注:

1.看此篇前你需要先知道springMvc的基本知识或者会用springMvc，可参考我的《springMvc的基本使用方式》

2.看此篇前你需要先看我的另一篇springMvc配置的文章:《三种情况的springMvc的IOC容器的创建过程》

3.相关的配置文件配置如下:

web.xml

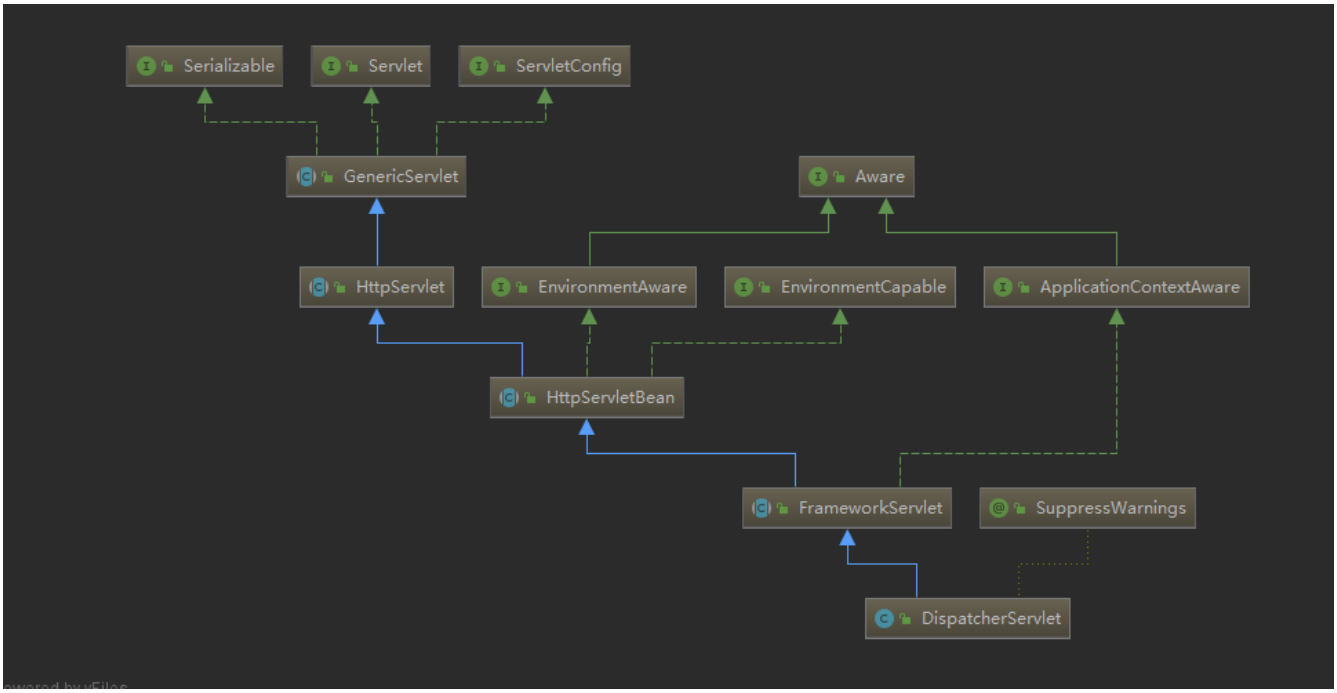
```
<servlet>
    <servlet-name >spring-mvc</servlet-name>
    <servlet-class >org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param >
        <param-name >contextConfigLocation</param-name>
        <param-value >classpath:/spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup >1</load-on-startup>
</servlet>
<servlet-mapping >
    <servlet-name >spring-mvc</servlet-name>
    <url-pattern >/</url-pattern>
</servlet-mapping>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

spring-mvc.xml

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value="/WEB-INF/views/"></property>
    <property name = "suffix" value = ".jsp"></property>
</bean>
```

整体分析

先看前端控制器DispatcherServlet的类继承关系图:



整个关系图来说不是很复杂，但是要注意FrameworkServlet与与httpServlet这两个类

分析前预备知识:springMvc9大组件的初始化

在《三种情况的springMvc的IOC容器的创建过程》这篇里我们提到了onRefresh()中进行了9大组件的初始化

```
//springMvc在工作的时候，关键位置都是由这些组件完成的
/** Well-known name for the MultipartResolver object in the bean factory for this namespace. */
//文件上传解析器,与文件上传有关
public static final String MULTIPART_RESOLVER_BEAN_NAME = "multipartResolver";

/** Well-known name for the LocaleResolver object in the bean factory for this namespace. */
//区域信息解析器,与国际化有关
public static final String LOCALE_RESOLVER_BEAN_NAME = "localeResolver";

/** Well-known name for the ThemeResolver object in the bean factory for this namespace. */
//主题解析器，支持主题系统更换
public static final String THEME_RESOLVER_BEAN_NAME = "themeResolver";

/**
 * Well-known name for the HandlerMapping object in the bean factory for this namespace.
 * Only used when "detectAllHandlerMappings" is turned off.
 * @see #setDetectAllHandlerMappings
 */
//handlerMapping
public static final String HANDLER_MAPPING_BEAN_NAME = "handlerMapping";

/**
 * Well-known name for the HandlerAdapter object in the bean factory for this namespace.
 * Only used when "detectAllHandlerAdapters" is turned off.
 * @see #setDetectAllHandlerAdapters
 */
//handlerMapping的适配器
public static final String HANDLER_ADAPTER_BEAN_NAME = "handlerAdapter";

/**
```

```
 * Well-known name for the HandlerExceptionResolver object in the bean factory for this namespace.
 * Only used when "detectAllHandlerExceptionResolvers" is turned off.
 * @see #setDetectAllHandlerExceptionResolvers
 */
//处理器异常解析器，支持强大的异常功能
public static final String HANDLER_EXCEPTION_RESOLVER_BEAN_NAME = "handlerExceptionResolver";

/**
 * Well-known name for the RequestToViewNameTranslator object in the bean factory for this
namespace.
 */
//视图名称转化器，就是如果处理方法没有返回值那么就将请求地址作为视图名
public static final String REQUEST_TO_VIEW_NAME_TRANSLATOR_BEAN_NAME = "viewNameTranslator";

/**
 * Well-known name for the ViewResolver object in the bean factory for this namespace.
 * Only used when "detectAllViewResolvers" is turned off.
 * @see #setDetectAllViewResolvers
 */
//视图解析
public static final String VIEW_RESOLVER_BEAN_NAME = "viewResolver";

/**
 * Well-known name for the FlashMapManager object in the bean factory for this namespace.
 */
//flashMap管理器，springMvc中运行重定向携带数据的功能(你没有看错就是重定向携带数据，不是转发携带数据)，具体的待查
public static final String FLASH_MAP_MANAGER_BEAN_NAME = "flashMapManager";

//SpringMvc九大组件保存的地方
//这9大组件全是接口，目的就是为了保证规范性，你可以自行实现自己的实现方式，但是需要尊从规范
private MultipartResolver multipartResolver;

private LocaleResolver localeResolver;

private ThemeResolver themeResolver;

private List<HandlerMapping> handlerMappings;

private List<HandlerAdapter> handlerAdapters;

private List<HandlerExceptionResolver> handlerExceptionResolvers;

private RequestToViewNameTranslator viewNameTranslator;

private FlashMapManager flashMapManager;

private List<ViewResolver> viewResolvers;
```

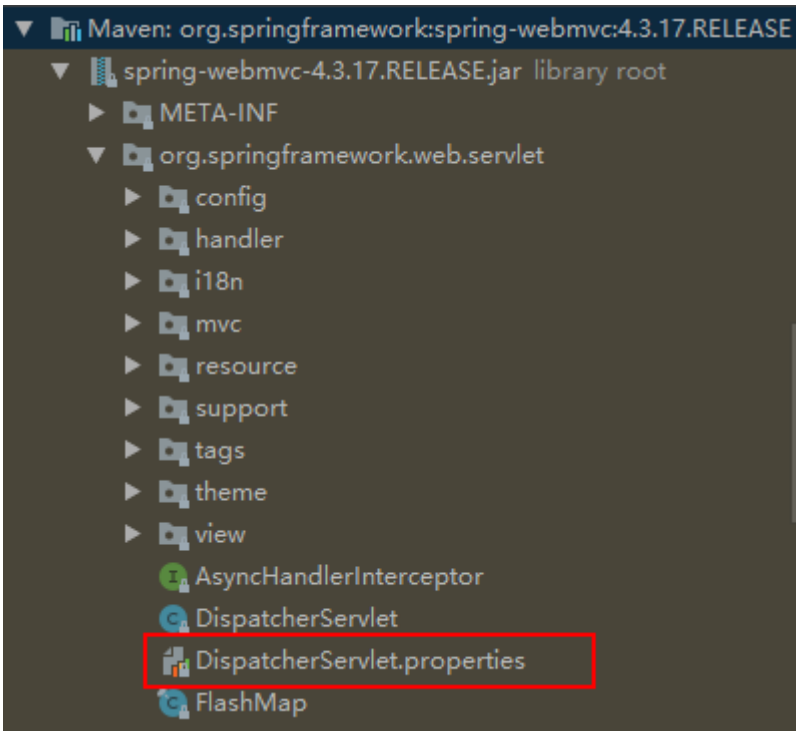
9大组件初始化:先在容器中寻找,如果找到则用容器中找到的(有些组件是使用类型找到,有些组件是使用ID找到)，如果没有找到就使用默认的配置

```
protected void onRefresh(ApplicationContext context) {
    initStrategies(context);
}
//初始化方法
protected void initStrategies(ApplicationContext context) {
```

```
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
    initRequestToViewNameTranslator(context);
    initViewResolvers(context);
    initFlashMapManager(context);
}
//分析其中几个组件的初始化
private void initHandlerMappings(ApplicationContext context) {
    this.handlerMappings = null;

    //detectAllHandlerMappings默认为true
    if (this.detectAllHandlerMappings) {
        // Find all HandlerMappings in the ApplicationContext, including ancestor contexts.
        //使用bean工厂找到所有的HandlerMapping类型
        Map<String, HandlerMapping> matchingBeans =
                BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class,
true, false);
        if (!matchingBeans.isEmpty()) {
            this.handlerMappings = new ArrayList<HandlerMapping>(matchingBeans.values());
            // We keep HandlerMappings in sorted order.
            AnnotationAwareOrderComparator.sort(this.handlerMappings);
        }
    }
    //detectAllHandlerMappings为false,此处可以这么改
    /**
    在wen.xml中
    <servlet>
        <servlet-name >spring-mvc</servlet-name>
        <servlet-class >org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param >
          <!-- 将detectAllHandlerMappings改为false -->
          <param-name >detectAllHandlerMappings</param-name>
          <param-value >false</param-value>
        </init-param>
        <init-param >
          <param-name >contextConfigLocation</param-name>
          <param-value >classpath:/spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup >1</load-on-startup>
    </servlet>
    */
    else {
        try {
            //在容器中找ID为handlerMapping的实例对象
            HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME, HandlerMapping.class);
            this.handlerMappings = Collections.singletonList(hm);
        }
        catch (NoSuchBeanDefinitionException ex) {
            // Ignore, we'll add a default HandlerMapping later.
        }
    }

    // Ensure we have at least one HandlerMapping, by registering
```

```
    // a default HandlerMapping if no other mappings are found.
    if (this.handlerMappings == null) {
        //如果都没找到则获取到默认的
        this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);
        if (logger.isDebugEnabled()) {
            logger.debug("No HandlerMappings found in servlet '" + getServletName() + "': using
default");
        }
    }
}
//获取默认的HandlerMapping
protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T> strategyInterface)
{
    String key = strategyInterface.getName();
    //从DispatcherServlet.properties这个文件中拿到相应的值，这个defaultStrategies请看下面
    /**
    默认的值是:org.springframework.web.servlet.HandlerMapping=
        org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
        org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
    */
    String value = defaultStrategies.getProperty(key);
    if (value != null) {
        String[] classNames = StringUtils.commaDelimitedListToStringArray(value);
        List<T> strategies = new ArrayList<T>(classNames.length);
        for (String className : classNames) {
            try {
                Class<?> clazz = ClassUtils.forName(className,
DispatcherServlet.class.getClassLoader());
                //根据类的信息得到类的实例
                Object strategy = createDefaultStrategy(context, clazz);
                //加入进list中
                strategies.add((T) strategy);
            }
            catch (ClassNotFoundException ex) {
                throw new BeanInitializationException(
                        "Could not find DispatcherServlet's default strategy class [" + className
+
                                "] for interface [" + key + "]", ex);
            }
            catch (LinkageError err) {
                throw new BeanInitializationException(
                        "Error loading DispatcherServlet's default strategy class [" + className +
                        "] for interface [" + key + "]: problem with class file or
dependent class", err);
            }
        }
        return strategies;
    }
    else {
        return new LinkedList<T>();
    }
}

private static final String DEFAULT_STRATEGIES_PATH = "DispatcherServlet.properties";

private static final Properties defaultStrategies;
```

```
static {
    // Load default strategy implementations from properties file.
    // This is currently strictly internal and not meant to be customized
    // by application developers.
    try {
        //拿到DispatcherServlet这个类路径下的资源，资源名为DispatcherServlet.properties
        ClassPathResource resource = new ClassPathResource(DEFAULT_STRATEGIES_PATH,
DispatcherServlet.class);
        //将DispatcherServlet.properties这个文件加载进来 DispatcherServlet.properties这个文件在
DispatcherServlet同一级目录
        //DispatcherServlet.properties的内容在下面
        defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);
    }
    catch (IOException ex) {
        throw new IllegalStateException("Could not load '" + DEFAULT_STRATEGIES_PATH + "': " +
ex.getMessage());
    }
}
```

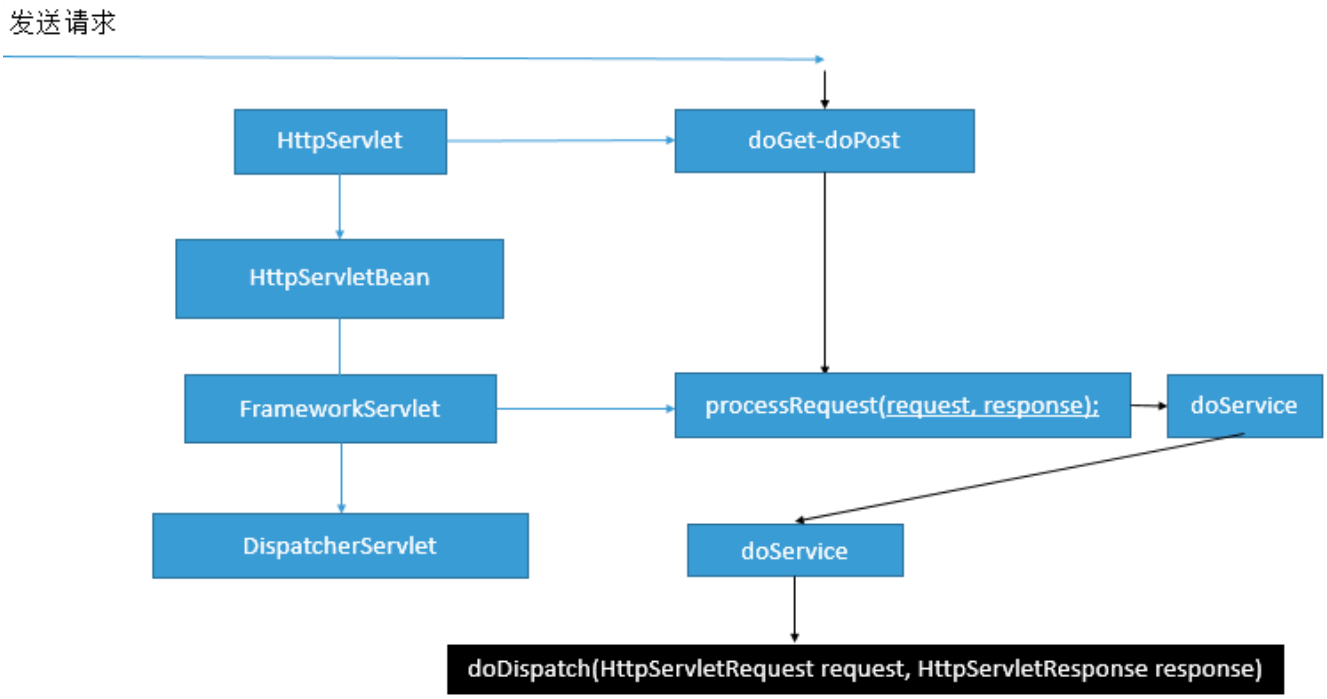


注:DispatcherServlet.properties的路径

```
DispatcherServlet.properties的内容:
# Default implementation classes for DispatcherServlet's strategy interfaces.
# Used as fallback when no matching beans are found in the DispatcherServlet context.
# Not meant to be customized by application developers.

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLo
caleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeReso
lver
```

```
org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrl
HandlerMapping,\
    org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHand
lerAdapter,\
    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annot
ation.AnnotationMethodHandlerExceptionResolver,\
    org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.D
efaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResource
ViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFla
shMapManager
```

整个运行的请求流程大致如下:



SSS

直接进入doDispatch()方法:

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;
    //如果有异步请求则拿到异步管理器
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
```

```
    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            //1.检查当前是否文件上传请求
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // Determine handler for the current request.
            //2.根据当前请求地址，找到可以处理的处理类
            mappedHandler = getHandler(processedRequest);
            //3.若没有找到相应的处理器则抛异常或者就到一个404页面
            if (mappedHandler == null || mappedHandler.getHandler() == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // Determine handler adapter for the current request.
            //4.拿到能执行这个处理器类中方法的适配器(其实就是一个反射工具:本例为
AnnotationMethodHandlerAdapter)
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

            // Process last-modified header, if supported by the handler.
            String method = request.getMethod();
            boolean isGet = "GET".equals(method);
            if (isGet || "HEAD".equals(method)) {
                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                if (logger.isDebugEnabled()) {
                    logger.debug("Last-Modified value for [" + getRequestUri(request) + "] is: " +
lastModified);
                }
                if (new ServletWebRequest(request, response).checkNotModified(lastModified) &&
isGet) {
                    return;
                }
            }

            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                return;
            }

            // Actually invoke the handler.
            //处理器(处理器)方法执行函数  控制器(Controller)也就是springMvc中的处理器(handler)
            //5.适配器执行目标方法,将目标方法的返回值作为视图名,设置并保存到ModelAndView中
            //目标方法无论怎么写,最终适配器执行完成后都会将执行完成后信息封装成ModelAndView
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }
            //如果没有视图名则设置一个默认的视图名
            applyDefaultViewName(processedRequest, mv);
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
```

```
                dispatchException = ex;
            }
            catch (Throwable err) {
                // As of 4.3, we're processing Errors thrown from handler methods as well,
                // making them available for @ExceptionHandler methods and other scenarios.
                dispatchException = new NestedServletException("Handler dispatch failed", err);
            }
            //转发到目标页面的
            //6.根据方法最终执行完成后封装的ModeelAndView，转发到对应页面而且ModeelAndView中的数据可以从请求域中获
得
            //其实也就是视图渲染过程 将域中的数据在页面展示；页面就是用来渲染模型数据的；
            processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
        }
        catch (Exception ex) {
            triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
        }
        catch (Throwable err) {
            triggerAfterCompletion(processedRequest, response, mappedHandler,
                    new NestedServletException("Handler processing failed", err));
        }
        finally {
            if (asyncManager.isConcurrentHandlingStarted()) {
                // Instead of postHandle and afterCompletion
                if (mappedHandler != null) {
                    mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
                }
            }
            else {
                // Clean up any resources used by a multipart request.
                if (multipartRequestParsed) {
                    cleanupMultipart(processedRequest);
                }
            }
        }
    }
```

1.所以请求过来DispatcherServlet收到请求

2.调用doDispatch()方法进行处理

1.getHandler()：根据当前请求地址找到能处理这个请求的目标处理器类(处理器)

根据当前请求在HandlerMapping中找到这个请求的映射信息,获取到目标处理器类

2.getHandlerAdapter()：根据当前处理器类获取到能执行这个处理器方法的适配器

根据当前处理器类,找到当前类的HandlerAdapter(处理器适配器)

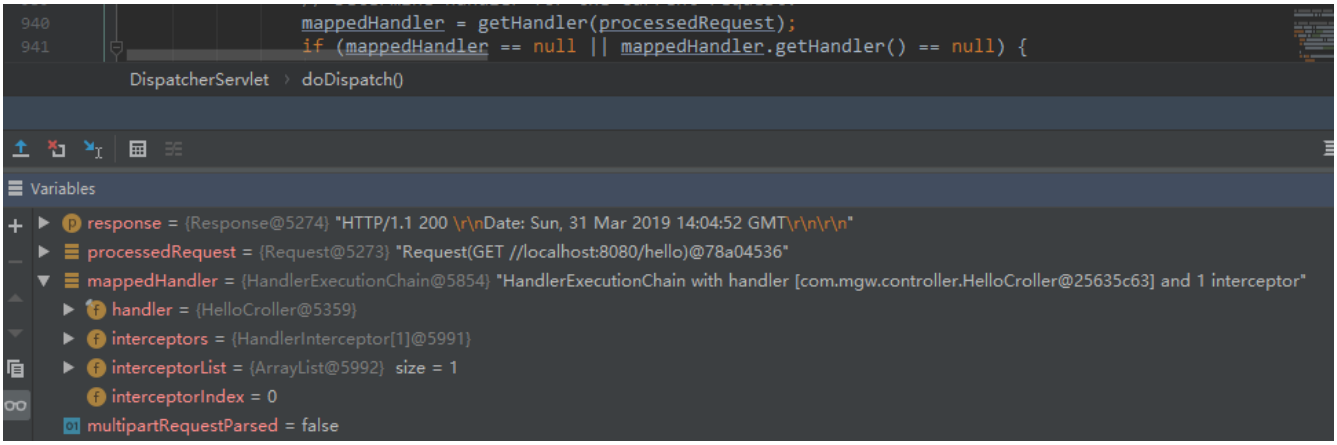
3.使用刚刚获取到的适配器(本例为AnnotationMethodHandlerAdapter)执行目标方法

4.目标方法执行后会返回一个ModelAndView对象

5.根据ModelAndView的信息转发到具体的页面,并可以在请求域中取出ModelAndView中的模型数据

方法细究:

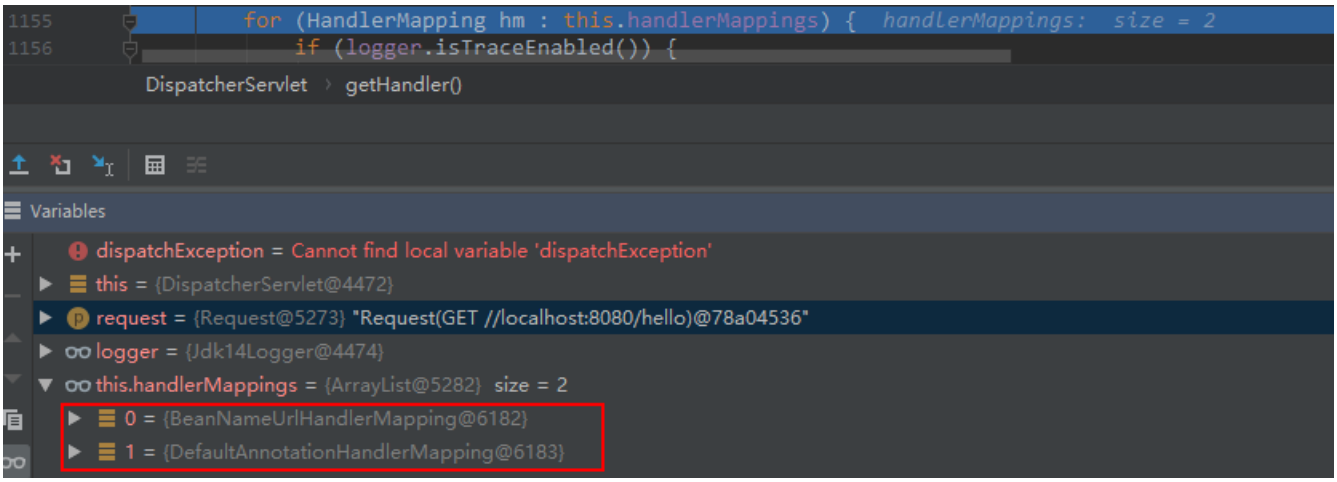
1.getHandler()的细究



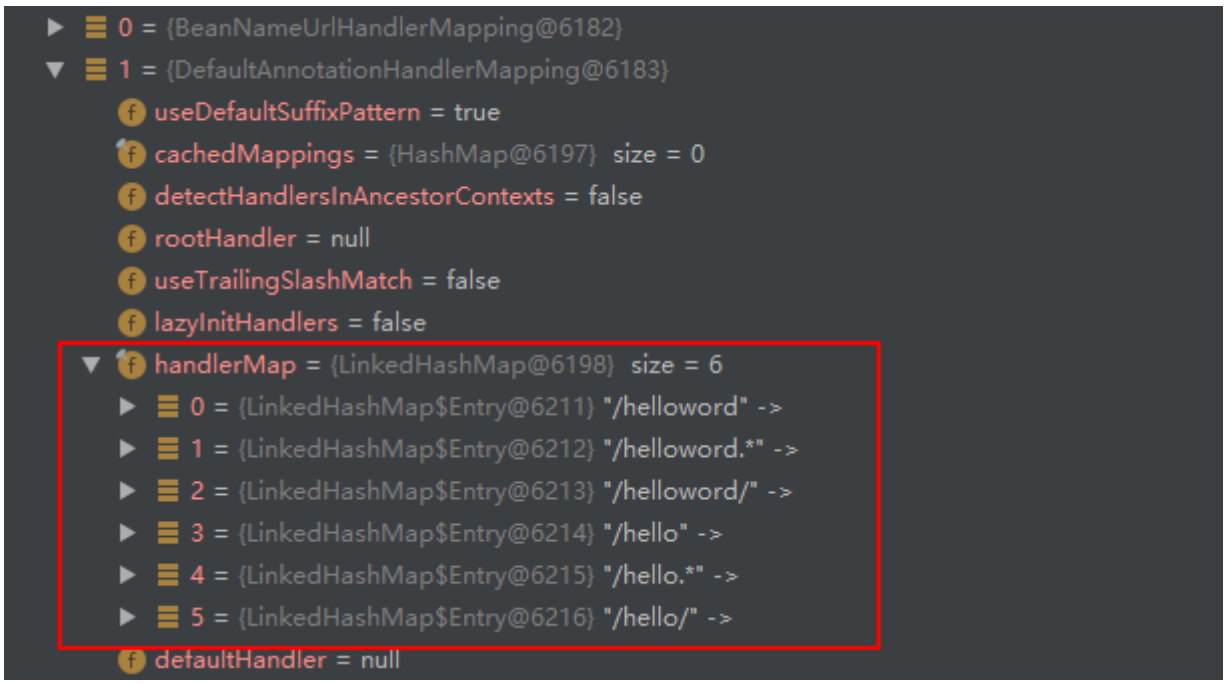
此方法的返回值为HandlerExecutionChain的对象mappedHandler，这是一个处理器执行链，这里面有处理器，拦截器等。

```
//找目标处理器
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    //遍历整个处理器映射遍历整个处理器映射(HandlerMapping)
    //HandlerMapping：处理器映射它里面保存了每一个处理器能处理那些请求的映射信息
    //为何handlerMapping中会有相应的handlerMap值呢?
    //因为:容器启动时创建Controller对象时扫描每一个处理器都能处理什么请求，保存到HandlerMapping的handlerMap属
性中，下次请求来时就可以匹配了
    for (HandlerMapping hm : this.handlerMappings) {
        if (logger.isTraceEnabled()) {
            logger.trace(
                    "Testing handler map [" + hm + "] in DispatcherServlet with name '" +
getServletName() + "'");
        }
        HandlerExecutionChain handler = hm.getHandler(request);
        if (handler != null) {
            return handler;
        }
    }
    return null;
}

@Override
public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    //得到处理器
    Object handler = getHandlerInternal(request);
    if (handler == null) {
        handler = getDefaultHandler();
    }
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?
    if (handler instanceof String) {
        String handlerName = (String) handler;
        handler = getApplicationContext().getBean(handlerName);
    }
    //组合成最后的处理器链
    HandlerExecutionChain executionChain = getHandlerExecutionChain(handler, request);
    if (CorsUtils.isCorsRequest(request)) {
        CorsConfiguration globalConfig =
this.globalCorsConfigSource.getCorsConfiguration(request);
```

```
        CorsConfiguration handlerConfig = getCorsConfiguration(handler, request);
        CorsConfiguration config = (globalConfig != null ? globalConfig.combine(handlerConfig) :
handlerConfig);
        executionChain = getCorsHandlerExecutionChain(request, executionChain, config);
    }
    return executionChain;
}
```



注:此例中的两个HandlerMapping

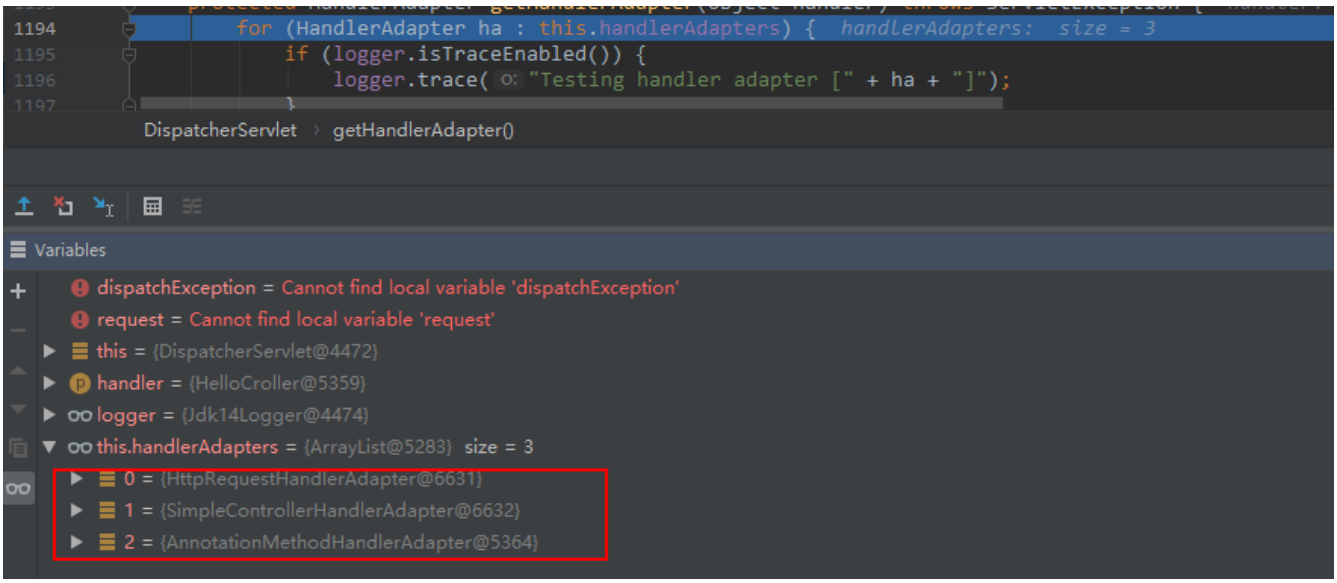


注:此例中实际使用的是DefaultAnnotationHandlerMapping这种类型的处理器映射

2.getHandlerAdapter()细究

```
//找目标处理器的适配器
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    //遍历所有的处理器适配器
    for (HandlerAdapter ha : this.handlerAdapters) {
        if (logger.isTraceEnabled()) {
            logger.trace("Testing handler adapter [" + ha + "]");
```

```
        }
        if (ha.supports(handler)) {
            return ha;
        }
    }
    throw new ServletException("No adapter for handler [" + handler +
            "]: The DispatcherServlet configuration needs to include a HandlerAdapter that
supports this handler");
}
//每一个处理器适配器的supports()方法都不同 当运行到第三个处理器适配器时
public boolean supports(Object handler) {
    return getMethodResolver(handler).hasHandlerMethods();
}
//得到方法解析器
private ServletHandlerMethodResolver getMethodResolver(Object handler) {
    //拿到处理器类就是我们自己写的那个Controller类
    Class<?> handlerClass = ClassUtils.getUserClass(handler);
    //那到处理器方法解析器
    ServletHandlerMethodResolver resolver = this.methodResolverCache.get(handlerClass);
    if (resolver == null) {
        synchronized (this.methodResolverCache) {
            resolver = this.methodResolverCache.get(handlerClass);
            if (resolver == null) {
                resolver = new ServletHandlerMethodResolver(handlerClass);
                this.methodResolverCache.put(handlerClass, resolver);
            }
        }
    }
    return resolver;
}
//判断是否有处理方法
public final boolean hasHandlerMethods() {
    //其实就是做了一个简单的处理器类中是否有方法的判断，因为之前拿到处理器时就已经知道了是否可解析本次请求,如果不能肯
定不会走到这一步，所以此处可以这么简单判断
    return !this.handlerMethods.isEmpty();
}
```



注:本例中的三个处理器适配器，我们实际可以使用的是AnnotationMethodHandlerAdapter这种适配器

3.handle()方法细究(AnnotationMethodHandlerAdapter这种适配器其实是过时,springMvc其实是推荐RequestMappingHandlerAdapter的，我暂时使用AnnotationMethodHandlerAdapter讲解，会单独讲RequestMappingHandlerAdapter这个解析器 祥见本篇的RequestMappingHandlerAdapter详解)

细究前的准备代码:

```
//使用@ModelAttribute注解保存一组数据
@ModelAttribute
public void hahaMyModelAttribute(Map<String, Object> map){

    Book book = new Book(100, "西游记", "吴承恩", 98, 10, 98.98);
    map.put("haha", book);
    System.out.println("modelAttribute方法...查询了图书并给你保存起来了...他用的map的类
型: "+map.getClass());
}
//updateBook方法作为主要的方法在后面的源码中进行相关的剖析
@RequestMapping("/updateBook")
public String updateBook(@RequestParam(value="author")String author,Map<String, Object> model,
                    HttpServletRequest request, @ModelAttribute("haha")Book book){
    //TODO
}
```

```
public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object
handler)
            throws Exception {

    Class<?> clazz = ClassUtils.getUserClass(handler);
    //此类是否标注了@SessionAttributes的注解
    Boolean annotatedWithSessionAttributes = this.sessionAnnotatedClassesCache.get(clazz);
    //没有标注时
    if (annotatedWithSessionAttributes == null) {
        //使用注解工具找@SessionAttributes的注解并缓存
        annotatedWithSessionAttributes = (AnnotationUtils.findAnnotation(clazz,
SessionAttributes.class) != null);
        this.sessionAnnotatedClassesCache.put(clazz, annotatedWithSessionAttributes);
    }

    //如果有@SessionAttributes的注解信息了
    if (annotatedWithSessionAttributes) {
        //做相关的检查和准备
        checkAndPrepare(request, response, this.cacheSecondsForSessionAttributeHandlers, true);
    }
    else {
        checkAndPrepare(request, response, true);
    }

    // Execute invokeHandlerMethod in synchronized block if required.
    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                return invokeHandlerMethod(request, response, handler);
```

```
            }
        }
    }
    //重点方法  执行目标方法
    return invokeHandlerMethod(request, response, handler);
}
//重点的invokeHandlerMethod()方法
protected ModelAndView invokeHandlerMethod(HttpServletRequest request, HttpServletResponse
response, Object handler)
            throws Exception {
    //拿到方法解析器
    ServletHandlerMethodResolver methodResolver = getMethodResolver(handler);
    //根据当前请求使用解析器解析得到使用哪一个处理方法
    Method handlerMethod = methodResolver.resolveHandlerMethod(request);
    //new 一个方法执行器
    ServletHandlerMethodInvoker methodInvoker = new ServletHandlerMethodInvoker(methodResolver);
    //将原生的request,response包装成一个对象，留待后用
    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    // 重点的地方:此处这个就是咱们经常使用的Map传值对象 创建一个BindingAwareModelMap对象  称为隐含模型
    ExtendedModelMap implicitModel = new BindingAwareModelMap();

    //重点方法:真正执行目标方法
    //目标方法执行期间确定参数值,提前执行@ModelAttribute等所有操作均在此
    //注意这个方法的传参
    Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler, webRequest,
implicitModel);
    //重点方法:将结果组合成一个ModelAndView对象
    ModelAndView mav =
            methodInvoker.getModelAndView(handlerMethod, handler.getClass(), result,
implicitModel, webRequest);
    methodInvoker.updateModelAttributes(handler, (mav != null ? mav.getModel() : null),
implicitModel, webRequest);
    //返回这个组合的ModelAndView对象
    return mav;
}
//重点的ServletHandlerMethodInvoker.invokeHandlerMethod()方法
//这个方法中重要的是反射过程中参数的确定很重要，为什么呢?
//因为springMvc的传参真的很随意，没有固定的数量  没有固定的方式 所以确定起来很困难 所以需要重点关注
public final Object invokeHandlerMethod(Method handlerMethod, Object handler,
            NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws Exception {

    /**
    public String updateBook(@RequestParam(value="author")String author,Map<String, Object> model,
                    HttpServletRequest request, @ModelAttribute("haha")Book book)
    需要执行的目标方法的全写
    public java.lang.String com.mgw.controller.updateBook(
            java.lang.String,
            java.util.Map,
            javax.servlet.http.HttpServletRequest,
            com.mgw.bean.Book)
    */
    Method handlerMethodToInvoke = BridgeMethodResolver.findBridgedMethod(handlerMethod);
    try {
        boolean debug = logger.isDebugEnabled();
        //拿到@SessionAttribute的所有相关信息并遍历
        for (String attrName : this.methodResolver.getActualSessionAttributeNames()) {
            //从sessionAttributeStore查询出相关的session信息
```

```
            Object attrValue = this.sessionAttributeStore.retrieveAttribute(webRequest, attrName);
            if (attrValue != null) {
                //将查出来的session信息信息放入隐含模型中
                implicitModel.addAttribute(attrName, attrValue);
            }
        }
        //获取标注了所有标注了@ModelAttribute注解的方法
        //本例中就是准备代码的public void hahaMyModelAttribute(Map<String, Object> map)这个方法
        for (Method attributeMethod : this.methodResolver.getModelAttributeMethods()) {
            //获取标注了@ModelAttribute注解的方法
            Method attributeMethodToInvoke =
BridgeMethodResolver.findBridgedMethod(attributeMethod);
            //重点方法:解析处理器的参数  注意传参  先确定ModelAttribute方法执行时需要使用的每一个参数的值
            Object[] args = resolveHandlerArguments(attributeMethodToInvoke, handler, webRequest,
implicitModel);
            if (debug) {
                logger.debug("Invoking model attribute method: " + attributeMethodToInvoke);
            }
            //找到当前@ModelAttribute注解的value值 见"辅助分析1"
            String attrName = AnnotationUtils.findAnnotation(attributeMethod,
ModelAttribute.class).value();
            //如果为value值不为""并且隐含模型中已经有了 直接跳过
            if (!"".equals(attrName) && implicitModel.containsAttribute(attrName)) {
                continue;
            }
            //将这个方法变为可访问的  反射的知识
            ReflectionUtils.makeAccessible(attributeMethodToInvoke);
            //反射执行标注了@ModelAttribute注解的方法  提前运行
            Object attrValue = attributeMethodToInvoke.invoke(handler, args);
            //当attrName为空字符串时
            if ("".equals(attrName)) {
                //拿到方法执行后的返回值类型
                Class<?> resolvedType =
GenericTypeResolver.resolveReturnType(attributeMethodToInvoke, handler.getClass());
                //为返回值类型起一个变量名并赋值给attrName 变量名为返回值类型首字母小写
                attrName = Conventions.getVariableNameForReturnType(attributeMethodToInvoke,
resolvedType, attrValue);
            }
            //如果当前隐含模型中没有相应的attrName值  将目标方法的返回值放入隐含模型
            //将提前运行的ModelAttribute方法放入隐含模型中
            if (!implicitModel.containsAttribute(attrName)) {
                implicitModel.addAttribute(attrName, attrValue);
            }
        }
        //再次调用resolveHandlerArguments()方法,解析目标方法的参数值
        /**
        本例:
            public String updateBook(@RequestParam(value="author")String author,Map<String,
Object> model,
                        HttpServletRequest request, @ModelAttribute("haha")Book book)
            全名称:
            public java.lang.String com.mgw.controller.updateBook(
            java.lang.String,
            java.util.Map,
            javax.servlet.http.HttpServletRequest,
            com.mgw.bean.Book)
        */
```

```
        Object[] args = resolveHandlerArguments(handlerMethodToInvoke, handler, webRequest,
implicitModel);
        if (debug) {
            logger.debug("Invoking request handler method: " + handlerMethodToInvoke);
        }
        //让目标方法变为可访问的
        ReflectionUtils.makeAccessible(handlerMethodToInvoke);
        //反射执行目标方法  现在知道了为啥springMvc中所有标注了@ModelAttribute注解的方法都一定运行在目标方法之
前了吧?
        return handlerMethodToInvoke.invoke(handler, args);
    }
    catch (IllegalStateException ex) {
        // Internal assertion failed (e.g. invalid signature):
        // throw exception with full handler method context...
        throw new HandlerMethodInvocationException(handlerMethodToInvoke, ex);
    }
    catch (InvocationTargetException ex) {
        // User-defined @ModelAttribute/@InitBinder/@RequestMapping method threw an exception...
        ReflectionUtils.rethrowException(ex.getTargetException());
        return null;
    }
}
//重点的resolveHandlerArguments()方法
//确定方法运行时使用到的每一个参数的值  大致总结见"辅助分析2"
private Object[] resolveHandlerArguments(Method handlerMethod, Object handler,
            NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws Exception {

    //拿到参数的类型
    Class<?>[] paramTypes = handlerMethod.getParameterTypes();
    //创建一个和参数个数一样多的数据,用来保存每一个参数的值
    Object[] args = new Object[paramTypes.length];

    for (int i = 0; i < args.length; i++) {
        //将第i个参数封装一下变为一个MethodParameter
        MethodParameter methodParam = new SynthesizingMethodParameter(handlerMethod, i);
        methodParam.initParameterNameDiscovery(this.parameterNameDiscoverer);
        GenericTypeResolver.resolveParameterType(methodParam, handler.getClass());
        String paramName = null;
        String headerName = null;
        boolean requestBodyFound = false;
        String cookieName = null;
        String pathVarName = null;
        String attrName = null;
        boolean required = false;
        String defaultValue = null;
        boolean validate = false;
        Object[] validationHints = null;
        //计数器注解找到了多少个
        int annotationsFound = 0;
        //找到当前参数的所有注解
        Annotation[] paramAnns = methodParam.getParameterAnnotations();

        //循环整个参数上的所有注解
        //找到所有注解，如果有注解就解析保存注解的信息
        for (Annotation paramAnn : paramAnns) {
            //注解为@RequestParam的情况
            if (RequestParam.class.isInstance(paramAnn)) {
```

```
                RequestParam requestParam = (RequestParam) paramAnn;
                //拿到@RequestParam注解中的name值
                paramName = requestParam.name();
                //拿到@RequestParam注解中的required值 表示这个是否是必须的
                required = requestParam.required();
                //拿到@RequestParam注解中的defaultValue值 表示如果没有就用磨人的这个值
                defaultValue = parseDefaultValueAttribute(requestParam.defaultValue());
                //annotationsFound计数器加1
                annotationsFound++;
            }
            //注解为@RequestHeader的情况
            else if (RequestHeader.class.isInstance(paramAnn)) {
                RequestHeader requestHeader = (RequestHeader) paramAnn;
                headerName = requestHeader.name();
                required = requestHeader.required();
                defaultValue = parseDefaultValueAttribute(requestHeader.defaultValue());
                annotationsFound++;
            }
            //注解为@RequestBody的情况
            else if (RequestBody.class.isInstance(paramAnn)) {
                requestBodyFound = true;
                annotationsFound++;
            }
            //注解为@CookieValue的情况
            else if (CookieValue.class.isInstance(paramAnn)) {
                CookieValue cookieValue = (CookieValue) paramAnn;
                cookieName = cookieValue.name();
                required = cookieValue.required();
                defaultValue = parseDefaultValueAttribute(cookieValue.defaultValue());
                annotationsFound++;
            }
            //注解为@PathVariable的情况
            else if (PathVariable.class.isInstance(paramAnn)) {
                PathVariable pathVar = (PathVariable) paramAnn;
                pathVarName = pathVar.value();
                annotationsFound++;
            }
            //注解为@ModelAttribute的情况
            else if (ModelAttribute.class.isInstance(paramAnn)) {
                ModelAttribute attr = (ModelAttribute) paramAnn;
                attrName = attr.value();
                annotationsFound++;
            }
            //注解为@Value的情况
            else if (Value.class.isInstance(paramAnn)) {
                defaultValue = ((Value) paramAnn).value();
            }
            else {
                Validated validatedAnn = AnnotationUtils.getAnnotation(paramAnn, Validated.class);
                if (validatedAnn != null ||
paramAnn.annotationType().getSimpleName().startsWith("Valid")) {
                    validate = true;
                    Object hints = (validatedAnn != null ? validatedAnn.value() :
AnnotationUtils.getValue(paramAnn));
                    validationHints = (hints instanceof Object[] ? (Object[]) hints : new Object[]
{hints});
                }
```

```
            }
        }
        //当annotationsFound计数器 > 1时 抛异常
        //其实就是上述注解只能标注一个
        if (annotationsFound > 1) {
            throw new IllegalStateException("Handler parameter annotations are exclusive choices -
" +
                    "do not specify more than one such annotation on the same parameter: " +
handlerMethod);
        }
        //没有找到注解时
        /**
        如果方法参数没有标注解:
            1.先看是否是普通参数 就是确定当前参数是否是原生API
            2.确定默认值类型
            3.确定Model或者Map型的参数  直接将之前new 的隐含模型给参数
        */
        if (annotationsFound == 0) {
            //解析普通参数
            // 解析普通参数 -> 解析自定义参数 -> 解析标准参数
            Object argValue = resolveCommonArgument(methodParam, webRequest);
            // 原生API解析成功就将值放入之前创建好的args数组中
            if (argValue != WebArgumentResolver.UNRESOLVED) {
                args[i] = argValue;
            }
            //是否有默认值  有默认值就解析默认值
            else if (defaultValue != null) {
                args[i] = resolveDefaultValue(defaultValue);
            }
            else {
                Class<?> paramType = methodParam.getParameterType();
                //如果参数类型是Model或者Map类型  为其确定值
                if (Model.class.isAssignableFrom(paramType) ||
Map.class.isAssignableFrom(paramType)) {
                    if (!paramType.isAssignableFrom(implicitModel.getClass())) {
                        throw new IllegalStateException("Argument [" + paramType.getSimpleName() +
"] is of type " +
                                "Model or Map but is not assignable from the actual model. You may
need to switch " +
                                "newer MVC infrastructure classes to use this argument.");
                    }
                    //将之前传入的隐含模型直接赋给参数
                    args[i] = implicitModel;
                }
                else if (SessionStatus.class.isAssignableFrom(paramType)) {
                    args[i] = this.sessionStatus;
                }
                else if (HttpEntity.class.isAssignableFrom(paramType)) {
                    args[i] = resolveHttpEntityRequest(methodParam, webRequest);
                }
                else if (Errors.class.isAssignableFrom(paramType)) {
                    throw new IllegalStateException("Errors/BindingResult argument declared " +
                            "without preceding model attribute. Check your handler method
signature!");
                }
                //确认参数是否是简单属性，简单属性是指 Integer，String等
                else if (BeanUtils.isSimpleProperty(paramType)) {
```

```
                    //是直接给空字符串
                    paramName = "";
                }
                else {
                    //自定义类型直接给attrName="" 目的是和@ModelAttribute注解拿中拿值时用同一个处理方法
                    attrName = "";
                }
            }
        }
        //确定值的环节
        if (paramName != null) {
            //解析RequestParam  这个paramName就是刚刚上面的@RequestParam注解中拿到的
            //或者是简单类型属性
            args[i] = resolveRequestParam(paramName, required, defaultValue, methodParam,
webRequest, handler);
        }
        else if (headerName != null) {
            args[i] = resolveRequestHeader(headerName, required, defaultValue, methodParam,
webRequest, handler);
        }
        else if (requestBodyFound) {
            args[i] = resolveRequestBody(methodParam, webRequest, handler);
        }
        else if (cookieName != null) {
            args[i] = resolveCookieValue(cookieName, required, defaultValue, methodParam,
webRequest, handler);
        }
        else if (pathVarName != null) {
            args[i] = resolvePathVariable(pathVarName, methodParam, webRequest, handler);
        }
        //解析参数上的@ModelAttribute注解 因为参数上也可能标注@ModelAttribute这个注解
        //或者确定自定义类型参数的值,还要将请求中的每一个参数的赋值到对象中
        else if (attrName != null) {
            //解析ModelAttribute 返回数据绑定器
            WebDataBinder binder =
                    resolveModelAttribute(attrName, methodParam, implicitModel, webRequest,
handler);
            boolean assignBindingResult = (args.length > i + 1 &&
Errors.class.isAssignableFrom(paramTypes[i + 1]));
            if (binder.getTarget() != null) {
                //重点方法 属性的绑定 将请求参数中提交的每一个值与java bean进行绑定 见辅助分析3
                doBind(binder, webRequest, validate, validationHints, !assignBindingResult);
            }
            args[i] = binder.getTarget();
            if (assignBindingResult) {
                args[i + 1] = binder.getBindingResult();
                i++;
            }
            implicitModel.putAll(binder.getBindingResult().getModel());
        }
    }

    return args;
}
//resolveCommonArgument() 解析普通参数
protected Object resolveCommonArgument(MethodParameter methodParameter, NativeWebRequest
webRequest)
```

```
            throws Exception {

    // Invoke custom argument resolvers if present...
    //自定义参数解析器时的情况
    if (this.customArgumentResolvers != null) {
        for (WebArgumentResolver argumentResolver : this.customArgumentResolvers) {
            Object value = argumentResolver.resolveArgument(methodParameter, webRequest);
            if (value != WebArgumentResolver.UNRESOLVED) {
                return value;
            }
        }
    }

    // Resolution of standard parameter types...
    //解析标准参数 就是原生API对象
    //拿到标准参数类型
    Class<?> paramType = methodParameter.getParameterType();
    //解析标准参数
    Object value = resolveStandardArgument(paramType, webRequest);
    if (value != WebArgumentResolver.UNRESOLVED && !ClassUtils.isAssignableValue(paramType,
value)) {
        throw new IllegalStateException("Standard argument type [" + paramType.getName() +
                "] resolved to incompatible value of type [" + (value != null ? value.getClass() :
null) +
                "]. Consider declaring the argument type in a less specific fashion.");
    }
    return value;
}
//resolveStandardArgument() 解析标准参数  确定参数是否是原生API
protected Object resolveStandardArgument(Class<?> parameterType, NativeWebRequest webRequest)
throws Exception {
    HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);
    HttpServletResponse response = webRequest.getNativeResponse(HttpServletResponse.class);

    //参数是否是Request
    if (ServletRequest.class.isAssignableFrom(parameterType) ||
            MultipartRequest.class.isAssignableFrom(parameterType)) {
        //拿出Request对象
        Object nativeRequest = webRequest.getNativeRequest(parameterType);
        if (nativeRequest == null) {
            throw new IllegalStateException(
                    "Current request is not of type [" + parameterType.getName() + "]: " +
request);
        }
        //将Request对象返回  这样你参数上就可以用了
        return nativeRequest;
    }
    //参数是否是Response
    else if (ServletResponse.class.isAssignableFrom(parameterType)) {
        this.responseArgumentUsed = true;
        Object nativeResponse = webRequest.getNativeResponse(parameterType);
        if (nativeResponse == null) {
            throw new IllegalStateException(
                    "Current response is not of type [" + parameterType.getName() + "]: " +
response);
        }
        return nativeResponse;
```

```
    }
    //参数是否是Session
    else if (HttpSession.class.isAssignableFrom(parameterType)) {
        return request.getSession();
    }
    //参数是否是Principal
    else if (Principal.class.isAssignableFrom(parameterType)) {
        return request.getUserPrincipal();
    }
    //参数是否是Locale
    else if (Locale.class == parameterType) {
        return RequestContextUtils.getLocale(request);
    }
    //参数是否是InputStream
    else if (InputStream.class.isAssignableFrom(parameterType)) {
        return request.getInputStream();
    }
    //参数是否是Reader
    else if (Reader.class.isAssignableFrom(parameterType)) {
        return request.getReader();
    }
    //参数是否是OutputStream
    else if (OutputStream.class.isAssignableFrom(parameterType)) {
        this.responseArgumentUsed = true;
        return response.getOutputStream();
    }
    //参数是否是Writer
    else if (Writer.class.isAssignableFrom(parameterType)) {
        this.responseArgumentUsed = true;
        return response.getWriter();
    }
    //都不是 返回一个UNRESOLVED的  其实就是Object UNRESOLVED = new Object(); 随便new了一个Object对象
    /**
    protected Object resolveStandardArgument(Class<?> parameterType,    NativeWebRequest
webRequest) throws Exception {
        if (WebRequest.class.isAssignableFrom(parameterType)) {
            return webRequest;
        }
        return WebArgumentResolver.UNRESOLVED;
    }
    */
    return super.resolveStandardArgument(parameterType, webRequest);
}
//解析resolveRequestParam()
private Object resolveRequestParam(String paramName, boolean required, String defaultValue,
            MethodParameter methodParam, NativeWebRequest webRequest, Object
handlerForInitBinderCall)
            throws Exception {
    //拿到参数的类型
    Class<?> paramType = methodParam.getParameterType();
    if (Map.class.isAssignableFrom(paramType) && paramName.length() == 0) {
        return resolveRequestParamMap((Class<? extends Map<?, ?>>) paramType, webRequest);
    }
    if (paramName.length() == 0) {
        paramName = getRequiredParameterName(methodParam);
    }
    Object paramValue = null;
```

```
    //判断是否和文件上传有关
    MultipartRequest multipartRequest = webRequest.getNativeRequest(MultipartRequest.class);
    if (multipartRequest != null) {
        List<MultipartFile> files = multipartRequest.getFiles(paramName);
        if (!files.isEmpty()) {
            paramValue = (files.size() == 1 ? files.get(0) : files);
        }
    }
    if (paramValue == null) {
        //调用servlet的request.getParameterValues(paramName)方法
        String[] paramValues = webRequest.getParameterValues(paramName);
        if (paramValues != null) {
            //如果长度为1则取第一个 否则直接取整个数组
            paramValue = (paramValues.length == 1 ? paramValues[0] : paramValues);
        }
    }
    //如果paramValue还是为null
    if (paramValue == null) {
        //判断有无默认值就是我们的@RequestParam注解中的defaultValue的值
        if (defaultValue != null) {
            paramValue = resolveDefaultValue(defaultValue);
        }
        //判断是否为必须的值就是我们的@RequestParam注解中的required的值
        else if (required) {
            //如果是抛MissingServletRequestParameterException这个异常
            raiseMissingParameterException(paramName, paramType);
        }
        paramValue = checkValue(paramName, paramValue, paramType);
    }
    //创建一个绑定器
    WebDataBinder binder = createBinder(webRequest, null, paramName);
    //绑定器初始化
    initBinder(handlerForInitBinderCall, paramName, binder, webRequest);
    return binder.convertIfNecessary(paramValue, paramType, methodParam);
}
//resolveModelAttribute()方法 解析ModelAttribute 返回数据绑定器
//确定自定义参数值或者ModelAttribute注解中拿值
private WebDataBinder resolveModelAttribute(String attrName, MethodParameter methodParam,
            ExtendedModelMap implicitModel, NativeWebRequest webRequest, Object handler) throws
Exception {

    // Bind request parameter onto object...
    String name = attrName;
    //如果attrName=""  就是我们上面说的 '自定义类型直接给attrName=""'
    if ("".equals(name)) {
        //如果是空串就将目标方法的参数类型小写作为值 例如: 参数为:Book oo ;实际name=book
        name = Conventions.getVariableNameForParameter(methodParam);
    }
    Class<?> paramType = methodParam.getParameterType();
    Object bindObject; //确定目标对象的值
    //如果隐含模型中有key指定的值(key为如果ModelAttribute注解标了value就是value的值，没标就是参数类型首字母小
写)
    if (implicitModel.containsKey(name)) {
        //直接拿到隐含模型中的对象
        bindObject = implicitModel.get(name);
    }
    //判断参数SessionAttribute标注的属性则从session中拿
```

```
    else if (this.methodResolver.isSessionAttribute(name, paramType)) {
        bindObject = this.sessionAttributeStore.retrieveAttribute(webRequest, name);
        if (bindObject == null) {
            //session中拿不到则抛异常
            raiseSessionRequiredException("Session attribute '" + name + "' required - not found
in session");
        }
    }
    //利用BeanUtils工具反射创建一个对象
    else {
        bindObject = BeanUtils.instantiateClass(paramType);
    }
    //创建一个绑定器
    WebDataBinder binder = createBinder(webRequest, bindObject, name);
    //重点方法 绑定器初始化
    initBinder(handler, name, binder, webRequest);
    return binder;
}
//initBinder() 绑定器初始化
protected void initBinder(Object handler, String attrName, WebDataBinder binder, NativeWebRequest
webRequest)
            throws Exception {

    if (this.bindingInitializer != null) {
        //统一调用bindingInitializer的initBinder()方法
        this.bindingInitializer.initBinder(binder, webRequest);
    }
    if (handler != null) {
        Set<Method> initBinderMethods = this.methodResolver.getInitBinderMethods();
        if (!initBinderMethods.isEmpty()) {
            boolean debug = logger.isDebugEnabled();
            for (Method initBinderMethod : initBinderMethods) {
                Method methodToInvoke = BridgeMethodResolver.findBridgedMethod(initBinderMethod);
                String[] targetNames = AnnotationUtils.findAnnotation(initBinderMethod,
InitBinder.class).value();
                if (targetNames.length == 0 || Arrays.asList(targetNames).contains(attrName)) {
                    Object[] initBinderArgs =
                            resolveInitBinderArguments(handler, methodToInvoke, binder,
webRequest);
                    if (debug) {
                        logger.debug("Invoking init-binder method: " + methodToInvoke);
                    }
                    ReflectionUtils.makeAccessible(methodToInvoke);
                    Object returnValue = methodToInvoke.invoke(handler, initBinderArgs);
                    if (returnValue != null) {
                        throw new IllegalStateException(
                                "InitBinder methods must not have a return value: " +
methodToInvoke);
                    }
                }
            }
        }
    }
}
//bindingInitializer的initBinder()方法 熟悉数据绑定的应该能猜到这个里面放啥了吧?
public void initBinder(WebDataBinder binder, WebRequest request) {
    binder.setAutoGrowNestedPaths(this.autoGrowNestedPaths);
```

```
    if (this.directFieldAccess) {
        binder.initDirectFieldAccess();
    }
    if (this.messageCodesResolver != null) {
        binder.setMessageCodesResolver(this.messageCodesResolver);
    }
    if (this.bindingErrorProcessor != null) {
        binder.setBindingErrorProcessor(this.bindingErrorProcessor);
    }
    if (this.validator != null && binder.getTarget() != null &&
            this.validator.supports(binder.getTarget().getClass())) {
        //绑定器中放入validator校验器
        binder.setValidator(this.validator);
    }
    if (this.conversionService != null) {
        //绑定器中放入conversionService
        binder.setConversionService(this.conversionService);
    }
    if (this.propertyEditorRegistrars != null) {
        for (PropertyEditorRegistrar propertyEditorRegistrar : this.propertyEditorRegistrars) {
            propertyEditorRegistrar.registerCustomEditors(binder);
        }
    }
}
//getModelAndView()方法  将结果组合成为ModelAndView对象
public ModelAndView getModelAndView(Method handlerMethod, Class<?> handlerType, Object
returnValue,
                ExtendedModelMap implicitModel, ServletWebRequest webRequest) throws Exception {

    ResponseStatus responseStatus = AnnotatedElementUtils.findMergedAnnotation(handlerMethod,
ResponseStatus.class);
    if (responseStatus != null) {
        HttpStatus statusCode = responseStatus.code();
        String reason = responseStatus.reason();
        if (!StringUtils.hasText(reason)) {
            webRequest.getResponse().setStatus(statusCode.value());
        }
        else {
            webRequest.getResponse().sendError(statusCode.value(), reason);
        }

        // to be picked up by the RedirectView
        webRequest.getRequest().setAttribute(View.RESPONSE_STATUS_ATTRIBUTE, statusCode);

        this.responseArgumentUsed = true;
    }

    // Invoke custom resolvers if present...
    if (customModelAndViewResolvers != null) {
        for (ModelAndViewResolver mavResolver : customModelAndViewResolvers) {
            ModelAndView mav = mavResolver.resolveModelAndView(
                    handlerMethod, handlerType, returnValue, implicitModel, webRequest);
            if (mav != ModelAndViewResolver.UNRESOLVED) {
                return mav;
            }
        }
    }
```

```
    if (returnValue instanceof HttpEntity) {
        handleHttpEntityResponse((HttpEntity<?>) returnValue, webRequest);
        return null;
    }
    else if (AnnotationUtils.findAnnotation(handlerMethod, ResponseBody.class) != null) {
        handleResponseBody(returnValue, webRequest);
        return null;
    }
    else if (returnValue instanceof ModelAndView) {
        ModelAndView mav = (ModelAndView) returnValue;
        mav.getModelMap().mergeAttributes(implicitModel);
        return mav;
    }
    else if (returnValue instanceof Model) {
        return new ModelAndView().addAllObjects(implicitModel).addAllObjects(((Model)
returnValue).asMap());
    }
    else if (returnValue instanceof View) {
        return new ModelAndView((View) returnValue).addAllObjects(implicitModel);
    }
    else if (AnnotationUtils.findAnnotation(handlerMethod, ModelAttribute.class) != null) {
        addReturnValueAsModelAttribute(handlerMethod, handlerType, returnValue, implicitModel);
        return new ModelAndView().addAllObjects(implicitModel);
    }
    else if (returnValue instanceof Map) {
        return new ModelAndView().addAllObjects(implicitModel).addAllObjects((Map<String, ?>)
returnValue);
    }
    else if (returnValue instanceof String) {
        return new ModelAndView((String) returnValue).addAllObjects(implicitModel);
    }
    else if (returnValue == null) {
        // Either returned null or was 'void' return.
        if (this.responseArgumentUsed || webRequest.isNotModified()) {
            return null;
        }
        else {
            // Assuming view name translation...
            return new ModelAndView().addAllObjects(implicitModel);
        }
    }
    else if (!BeanUtils.isSimpleProperty(returnValue.getClass())) {
        // Assume a single model attribute...
        addReturnValueAsModelAttribute(handlerMethod, handlerType, returnValue, implicitModel);
        return new ModelAndView().addAllObjects(implicitModel);
    }
    else {
        throw new IllegalArgumentException("Invalid handler method return value: " + returnValue);
    }
}
```

辅助分析1:

```
方法如果标注了@ModelAttribute的value值  attrName = value标注的值
eg:
    @ModelAttribute(value="aaa")
    public void hahaMyModelAttribute(Map<String, Object> map) {}
    则attrName = aaa
当没标时 attrName = 方法返回值类型首字母小写 比如void , person

@ModelAttribute的另一个作用:可以把目标方法运行后的返回值按指定的attrName为key,返回值为value的形式放入隐含模型
eg:
        public void hahaMyModelAttribute1(Map<String, Object> map) {}  则对应的k-v为:(void,null)
```

辅助分析2:

辅助分析2:

```
如何为参数确定值?
        参数标了注解:
                保存是那个注解的详细信息，最后确定值
                如果参数有ModelAttribute注解
                        拿到ModelAttribute注解的值让attrName保存
                        attrName="xxx值"
        参数没有标注解:
                1.先看是否是普通参数(原生API)
                2.再看是否是Model或者Map 传入隐含模型
                3.确认自定义类型的参数没有ModelAttribute注解
                        1.先看是否是原生API
                        2.再看是否是Model或者Map
                        3.再看是否是其他类型  比如:SessionStatus,HttpEntity，Errors等
                        4.再看是否是简单属性
                                是paramName = ""直接给空串
                        5.直接给attrName=""  这实际上就是为自定义对象做解析前的准备
如果是自定义类型对象，最终产生两个效果:
        1.如果这个参数标注了ModelAttribute注解就给attrName赋值为这个注解的value值(attrName="xxx值")
        2.如果这个参数没有标注了ModelAttribute注解就给attrName赋值为空串(attrName="")

自定义类型对象(POJO)的值的确认:
        1.如果隐藏模型中有这个key(标注了ModelAttribute注解就是注解指定的value,没有就是参数类型的首字母小写)指定的
值
        2.如果是SessionAttribute标注的属性,就从session中拿
        3.如果都不是就利用反射创建对象
```

辅助分析3:

```
//数据绑定
private void doBind(WebDataBinder binder, NativeWebRequest webRequest, boolean validate,
                        Object[] validationHints, boolean failOnErrors) throws Exception {
        //做数据绑定
        doBind(binder, webRequest);
    //数据校验
        if (validate) {
                binder.validate(validationHints);
        }
        if (failOnErrors && binder.getBindingResult().hasErrors()) {
        //如果校验出现问题 错误结果放进BindingResult中
```

```
                throw new BindException(binder.getBindingResult());
        }
}
//doBind()方法
protected void doBind(WebDataBinder binder, NativeWebRequest webRequest) throws Exception {
    ServletRequestDataBinder servletBinder = (ServletRequestDataBinder) binder;
    //绑定数据
    servletBinder.bind(webRequest.getNativeRequest(ServletRequest.class));
}
//bind () 方法
public void bind(ServletRequest request) {
    //将request中的数据封装到PropertyValues中便于取数据  就是将带来的参数重封装
    MutablePropertyValues mpvs = new ServletRequestParameterPropertyValues(request);
    MultipartRequest multipartRequest = WebUtils.getNativeRequest(request, MultipartRequest.class);
    //如果是多媒体(例如上传功能)就绑定多媒体相关的数据
    if (multipartRequest != null) {
        bindMultipart(multipartRequest.getMultiFileMap(), mpvs);
    }
    //添加绑定数据  如果子类需要 子类重写
    addBindValues(mpvs, request);
    //做绑定
    doBind(mpvs);
}

//doBind() 方法  做绑定
protected void doBind(MutablePropertyValues mpvs) {
    //做字段相关的检查 比如字段名匹配否?
    checkFieldDefaults(mpvs);
    checkFieldMarkers(mpvs);
    super.doBind(mpvs);
}
//父类的doBind()方法
protected void doBind(MutablePropertyValues mpvs) {
    //也是字段相关的检查等
    checkAllowedFields(mpvs);
    checkRequiredFields(mpvs);
    //应用属性值 其实就是bean属性值帮上传过来的值
    applyPropertyValues(mpvs);
}
//applyPropertyValues() 方法
protected void applyPropertyValues(MutablePropertyValues mpvs) {
    try {
        // Bind request parameters onto target object.
        //属性值与bean的绑定
        getPropertyAccessor().setPropertyValues(mpvs, isIgnoreUnknownFields(),
isIgnoreInvalidFields());
    }
    catch (PropertyBatchUpdateException ex) {
        // Use bind error processor to create FieldErrors.
        for (PropertyAccessException pae : ex.getPropertyAccessExceptions()) {
            getBindingErrorProcessor().processPropertyAccessException(pae,
getInternalBindingResult());
        }
    }
}
```

4.processDispatchResult()方法细究

```
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
                        HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception)
throws Exception {

    //首先说一下这个ModelAndView  这个不但有返回的视图名还有隐含模型的数据

        boolean errorView = false;

        if (exception != null) {
                if (exception instanceof ModelAndViewDefiningException) {
                        logger.debug("ModelAndViewDefiningException encountered", exception);
                        mv = ((ModelAndViewDefiningException) exception).getModelAndView();
                }
                else {
                        Object handler = (mappedHandler != null ? mappedHandler.getHandler() :
null);
                        mv = processHandlerException(request, response, handler, exception);
                        errorView = (mv != null);
                }
        }

        // Did the handler return a view to render?
        if (mv != null && !mv.wasCleared()) {
        //重点方法  渲染
                render(mv, request, response);
                if (errorView) {
                        WebUtils.clearErrorRequestAttributes(request);
                }
        }
        else {
                if (logger.isDebugEnabled()) {
                        logger.debug("Null ModelAndView returned to DispatcherServlet with name '" +
getServletName() +
                                        "': assuming HandlerAdapter completed request handling");
                }
        }

        if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
                // Concurrent handling started during a forward
                return;
        }

        if (mappedHandler != null) {
                mappedHandler.triggerAfterCompletion(request, response, null);
        }
}
//render() 渲染
protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response)
throws Exception {
        // Determine locale for request and apply it to the response.
        Locale locale = this.localeResolver.resolveLocale(request);
        response.setLocale(locale);

    /**
```

```
    此处说说View与ViewResolver
    ViewResolver是一个接口  里面只有一个方法  其作用就是传入视图名，返回视图(View)对象
        public interface ViewResolver {
                //传入视图名 返回视图对象
                View resolveViewName(String viewName, Locale locale) throws Exception;
        }
    下面会继续说
        */
        View view;
        if (mv.isReference()) {
                // We need to resolve the view name.
        //重点方法 通过名图名得到视图对象  注意传入的参数，第一个参数就是返回的视图名 第二个参数就是隐含模型
                view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);
                if (view == null) {
            //没有匹配的视图抛异常
                        throw new ServletException("Could not resolve view with name '" +
mv.getViewName() +
                                        "' in servlet with name '" + getServletName() + "'");
                }
        }
        else {
                // No need to lookup: the ModelAndView object contains the actual View object.
                view = mv.getView();
                if (view == null) {
                        throw new ServletException("ModelAndView [" + mv + "] neither contains a
view name nor a " +
                                        "View object in servlet with name '" + getServletName() +
"'");
                }
        }

        // Delegate to the View object for rendering.
        if (logger.isDebugEnabled()) {
                logger.debug("Rendering view [" + view + "] in DispatcherServlet with name '" +
getServletName() + "'");
        }
        try {
                if (mv.getStatus() != null) {
                        response.setStatus(mv.getStatus().value());
                }
        //调用View对象的render()方法  注意传入的参数 第一个参数为隐藏模型
                view.render(mv.getModelInternal(), request, response);
        }
        catch (Exception ex) {
                if (logger.isDebugEnabled()) {
                        logger.debug("Error rendering view [" + view + "] in DispatcherServlet with
name '" +
                                        getServletName() + "'", ex);
                }
                throw ex;
        }
}
//resolveViewName() 通过名图名得到视图对象
protected View resolveViewName(String viewName, Map<String, Object> model, Locale locale,
                        HttpServletRequest request) throws Exception {

    //遍历所有的视图解析器方法 找到合适的就返回 为什么说合适的就行？因为一种多种ViewResolver都可解析同一种视图
```

```
    /**
    返回View对象;
        视图解析器得到View对象的流程就是，所有配置的视图解析器都来尝试根据视图名（返回值）得到View（视图）对象；如果能得到就返回，得不到就换下一个视图解析器；例如我们可以自定义视图和视图解析器  我们自定义的本就不能解析所有视图 所以可能会返回null 此时继续循环找别的视图解析器继续
    */
        for (ViewResolver viewResolver : this.viewResolvers) {
                View view = viewResolver.resolveViewName(viewName, locale);
                if (view != null) {
                        return view;
                }
        }
        return null;
}
//InternalResourceViewResolver的resolveViewName()方法
public View resolveViewName(String viewName, Locale locale) throws Exception {
        if (!isCache()) {
                return createView(viewName, locale);
        }
        else {

                Object cacheKey = getCacheKey(viewName, locale);
        //优先从缓存中拿
                View view = this.viewAccessCache.get(cacheKey);
        //缓存中没有时
                if (view == null) {
                        synchronized (this.viewCreationCache) {
                                view = this.viewCreationCache.get(cacheKey);
                //此处两个(view == null)就是单例中的双重检测
                                if (view == null) {
                                        // Ask the subclass to create the View object.
                    //创建视图对象View  使用委派模式调用子类的方法
                                        view = createView(viewName, locale);
                                        if (view == null && this.cacheUnresolved) {
                                                view = UNRESOLVED_VIEW;
                                        }
                                        if (view != null) {
                                                this.viewAccessCache.put(cacheKey, view);
                        //将创建的视图对象加入缓存中
                                                this.viewCreationCache.put(cacheKey, view);
                                                if (logger.isTraceEnabled()) {
                                                        logger.trace("Cached view [" + cacheKey +
"]");
                                                }
                                        }
                                }
                        }
                }
                return (view != UNRESOLVED_VIEW ? view : null);
        }
}
//createView()方法 创建视图对象View
protected View createView(String viewName, Locale locale) throws Exception {
        // If this resolver is not supposed to handle the given view,
        // return null to pass on to the next resolver in the chain.
        if (!canHandle(viewName, locale)) {
                return null;
```

```
        }
        // Check for special "redirect:" prefix.
    //如果以"redirect:"开始的
        if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
                String redirectUrl = viewName.substring(REDIRECT_URL_PREFIX.length());
        //直接new一个RedirectView对象
                RedirectView view = new RedirectView(redirectUrl, isRedirectContextRelative(),
isRedirectHttp10Compatible());
                view.setHosts(getRedirectHosts());
                return applyLifecycleMethods(viewName, view);
        }
        // Check for special "forward:" prefix.
    ////如果以"forward:"开始的
        if (viewName.startsWith(FORWARD_URL_PREFIX)) {
        //转发的url地址
                String forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length());
        //new一个InternalResourceViewd对象
                return new InternalResourceView(forwardUrl);
        }
        // Else fall back to superclass implementation: calling loadView.
    //如果没有前缀就是用父类默认创建一个View
        return super.createView(viewName, locale);
}
//createView() 没有前缀时使用父类默认创建一个View
protected View createView(String viewName, Locale locale) throws Exception {
        //载入视图
    return loadView(viewName, locale);
}
//loadView() 载入视图
protected View loadView(String viewName, Locale locale) throws Exception {
    //组建视图
    AbstractUrlBasedView view = buildView(viewName);
    View result = applyLifecycleMethods(viewName, view);
    return (view.checkResource(locale) ? result : null);
}
//InternalResourceViewResolver类的buildView()
protected AbstractUrlBasedView buildView(String viewName) throws Exception {
    //先调用其父类的buildView()方法  强行使用InternalResourceView 实际上有子类JstlView时 它用的是其子类
        InternalResourceView view = (InternalResourceView) super.buildView(viewName);
        if (this.alwaysInclude != null) {
                view.setAlwaysInclude(this.alwaysInclude);
        }
        view.setPreventDispatchLoop(true);
        return view;
}
//父类的buildView()
protected AbstractUrlBasedView buildView(String viewName) throws Exception {
        AbstractUrlBasedView view = (AbstractUrlBasedView)
BeanUtils.instantiateClass(getViewClass());
        //url的拼串 getPrefix()就是我们在配置文件中配置的前缀  getSuffix()就是我们在配置文件中配置的后缀
    /**
    <property name = "prefix" value="/WEB-INF/views/"></property>
        <property name = "suffix" value = ".jsp"></property>
    */
    view.setUrl(getPrefix() + viewName + getSuffix());
        //下面就是设置各种属性
        String contentType = getContentType();
```

```
        if (contentType != null) {
                view.setContentType(contentType);
        }

        view.setRequestContextAttribute(getRequestContextAttribute());
        view.setAttributesMap(getAttributesMap());

        Boolean exposePathVariables = getExposePathVariables();
        if (exposePathVariables != null) {
                view.setExposePathVariables(exposePathVariables);
        }
        Boolean exposeContextBeansAsAttributes = getExposeContextBeansAsAttributes();
        if (exposeContextBeansAsAttributes != null) {
                view.setExposeContextBeansAsAttributes(exposeContextBeansAsAttributes);
        }
        String[] exposedContextBeanNames = getExposedContextBeanNames();
        if (exposedContextBeanNames != null) {
                view.setExposedContextBeanNames(exposedContextBeanNames);
        }

        return view;
}
//render()  调用返回的view的render()方法
public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)
throws Exception {
        if (logger.isTraceEnabled()) {
                logger.trace("Rendering view with name '" + this.beanName + "' with model " + model
+
                        " and static attributes " + this.staticAttributes);
        }
        //创建一个合并的输出模型
        Map<String, Object> mergedModel = createMergedOutputModel(model, request, response);
    //准备响应
        prepareResponse(request, response);
    //渲染要给页面输出的数据
        renderMergedOutputModel(mergedModel, getRequestToExpose(request), response);
}
//createMergedOutputModel()  创建一个合并的输出模型
protected Map<String, Object> createMergedOutputModel(Map<String, ?> model, HttpServletRequest
request,
                        HttpServletResponse response) {

        @SuppressWarnings("unchecked")
        Map<String, Object> pathVars = (this.exposePathVariables ?
                        (Map<String, Object>) request.getAttribute(View.PATH_VARIABLES) : null);

        // Consolidate static and dynamic model attributes.
        int size = this.staticAttributes.size();
        size += (model != null ? model.size() : 0);
        size += (pathVars != null ? pathVars.size() : 0);
        //创建一个合并的模型 主要是用一个统一的Map来存放给前端的各种数据
        Map<String, Object> mergedModel = new LinkedHashMap<String, Object>(size);
        mergedModel.putAll(this.staticAttributes);
        if (pathVars != null) {
                mergedModel.putAll(pathVars);
        }
        if (model != null) {
```

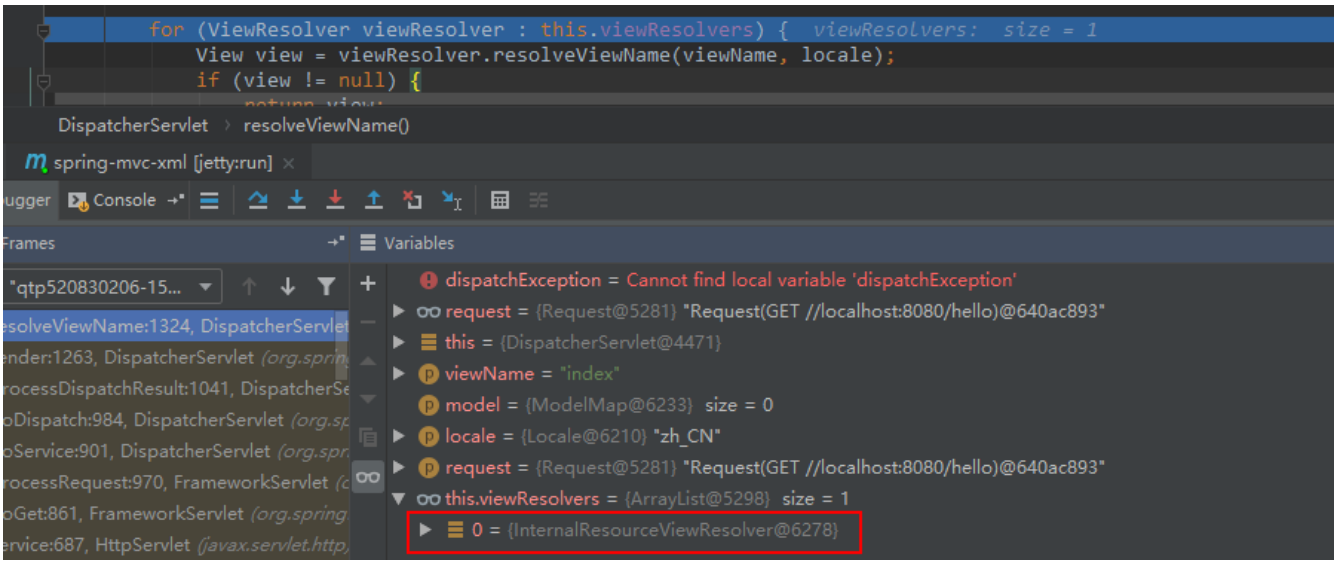
```
                mergedModel.putAll(model);
        }

        // Expose RequestContext?
        if (this.requestContextAttribute != null) {

        /**
        protected RequestContext createRequestContext(
                        HttpServletRequest request, HttpServletResponse response, Map<String,
Object> model) {
            //把模型数据放入RequestContext
            return new RequestContext(request, response, getServletContext(), model);
        }
        */
                mergedModel.put(this.requestContextAttribute, createRequestContext(request,
response, mergedModel));
        }

        return mergedModel;
}
//renderMergedOutputModel() 渲染要给页面输出的数据
protected void renderMergedOutputModel(
                        Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response) throws Exception {

        // Expose the model object as request attributes.
    //暴露模型数据到Request域中
    // 现在你应该知道为什么springMvc自定义(Map,Model,BindingAwareModelMap)的给页面输出的数据都可以在request域
中取到了吧?
        exposeModelAsRequestAttributes(model, request);

        // Expose helpers as request attributes, if any.
        exposeHelpers(request);

        // Determine the path for the request dispatcher.
    //请求转发的路径
        String dispatcherPath = prepareForRendering(request, response);

        // Obtain a RequestDispatcher for the target resource (typically a JSP).
    //拿到转发器
        RequestDispatcher rd = getRequestDispatcher(request, dispatcherPath);
        if (rd == null) {
                throw new ServletException("Could not get RequestDispatcher for [" + getUrl() +
                                "]: Check that the corresponding file exists within your web
application archive!");
        }

        // If already included or response already committed, perform include, else forward.
        if (useInclude(request, response)) {
                response.setContentType(getContentType());
                if (logger.isDebugEnabled()) {
                        logger.debug("Including resource [" + getUrl() + "] in InternalResourceView
'" + getBeanName() + "'");
                }
                rd.include(request, response);
        }
```

```
        else {
                // Note: The forwarded resource is supposed to determine the content type itself.
                if (logger.isDebugEnabled()) {
                        logger.debug("Forwarding to resource [" + getUrl() + "] in
InternalResourceView '" + getBeanName() + "'");
                }
        //转发到页面
                rd.forward(request, response);
        }
}
//exposeModelAsRequestAttributes ()  暴露模型数据到Request域中
protected void exposeModelAsRequestAttributes(Map<String, Object> model, HttpServletRequest request)
throws Exception {
        //循环取出Model中的数据
    for (Map.Entry<String, Object> entry : model.entrySet()) {
                String modelName = entry.getKey();
                Object modelValue = entry.getValue();
                if (modelValue != null) {
            //放入reuqest域
                        request.setAttribute(modelName, modelValue);
                        if (logger.isDebugEnabled()) {
                                logger.debug("Added model object '" + modelName + "' of type [" +
modelValue.getClass().getName() +
                                                "] to request in view with name '" + getBeanName() +
"'");
                        }
                }
                else {
                        request.removeAttribute(modelName);
                        if (logger.isDebugEnabled()) {
                                logger.debug("Removed model object '" + modelName +
                                                "' from request in view with name '" + getBeanName()
+ "'");
                        }
                }
        }
}
/**
此处继续说说View与ViewResolver
ViewResolver是一个接口  里面只有一个方法  其作用就是传入视图名，返回视图(View)对象
public interface ViewResolver {
        //传入视图名 返回视图对象
        View resolveViewName(String viewName, Locale locale) throws Exception;
}
public interface View {

        String RESPONSE_STATUS_ATTRIBUTE = View.class.getName() + ".responseStatus";

        String PATH_VARIABLES = View.class.getName() + ".pathVariables";

        String SELECTED_CONTENT_TYPE = View.class.getName() + ".selectedContentType";

        String getContentType();
        //渲染
        void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)
throws Exception;
}
```

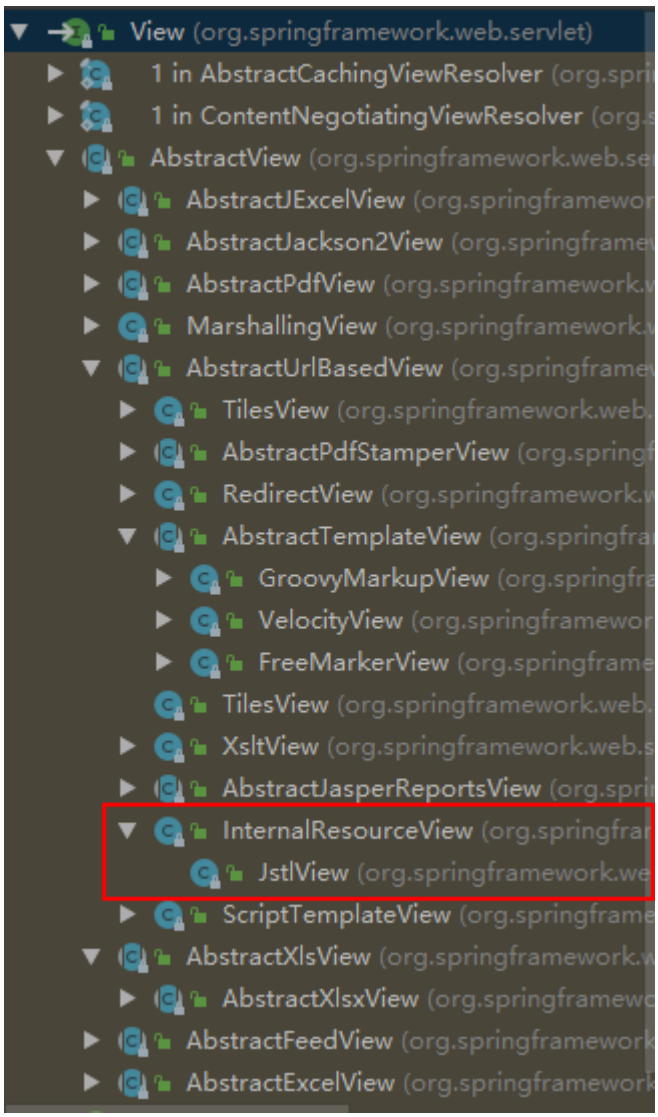
```
一句话:
视图解析器(ViewResolver)只是为了得到视图对象;
视图(View)对象才能真正的转发 (将模型数据全部放在请求域中) 或者重定向到页面
视图对象才能真正的渲染视图;
*/
//额外分析下RedirectView(重定向)和InternalResourceView(转发)  这两种视图
//RedirectView使用其父类AbstractView的render()方法  这个方法我们上面分析过了  只看下RedirectView重写的
renderMergedOutputModel()
public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)
throws Exception {
        if (logger.isTraceEnabled()) {
                logger.trace("Rendering view with name '" + this.beanName + "' with model " + model
+
                        " and static attributes " + this.staticAttributes);
        }

        Map<String, Object> mergedModel = createMergedOutputModel(model, request, response);
        prepareResponse(request, response);
        renderMergedOutputModel(mergedModel, getRequestToExpose(request), response);
}
//RedirectView重写的renderMergedOutputModel()
protected void renderMergedOutputModel(Map<String, Object> model, HttpServletRequest request,
                        HttpServletResponse response) throws IOException {

    //目标地址
        String targetUrl = createTargetUrl(model, request);
        targetUrl = updateTargetUrl(targetUrl, model, request, response);
        //FlashMap 还记得我们上面说的九大组件初始化哪里的那个flashMap吗? 没错这个FlashMap就是九大组件之一
    //给页面带一些数据
        FlashMap flashMap = RequestContextUtils.getOutputFlashMap(request);
        if (!CollectionUtils.isEmpty(flashMap)) {
                UriComponents uriComponents = UriComponentsBuilder.fromUriString(targetUrl).build();
                flashMap.setTargetRequestPath(uriComponents.getPath());
                flashMap.addTargetRequestParams(uriComponents.getQueryParams());
                FlashMapManager flashMapManager = RequestContextUtils.getFlashMapManager(request);
                if (flashMapManager == null) {
                        throw new IllegalStateException("FlashMapManager not found despite output
FlashMap having been set");
                }
                flashMapManager.saveOutputFlashMap(flashMap, request, response);
        }
        //重定向
        sendRedirect(request, response, targetUrl, this.http10Compatible);
}
//重定向
protected void sendRedirect(HttpServletRequest request, HttpServletResponse response,
                        String targetUrl, boolean http10Compatible) throws IOException {

    //URL
        String encodedURL = (isRemoteHost(targetUrl) ? targetUrl :
response.encodeRedirectURL(targetUrl));
        if (http10Compatible) {
                HttpStatus attributeStatusCode = (HttpStatus)
request.getAttribute(View.RESPONSE_STATUS_ATTRIBUTE);
                if (this.statusCode != null) {
            //设置状态码
                        response.setStatus(this.statusCode.value());
```

```
            //设置Location头
                        response.setHeader("Location", encodedURL);
                }
                else if (attributeStatusCode != null) {
                        response.setStatus(attributeStatusCode.value());
                        response.setHeader("Location", encodedURL);
                }
                else {
                        // Send status code 302 by default.
            //重定向
                        response.sendRedirect(encodedURL);
                }
        }
        else {
                HttpStatus statusCode = getHttp11StatusCode(request, response, targetUrl);
                response.setStatus(statusCode.value());
                response.setHeader("Location", encodedURL);
        }
}
//分析InternalResourceView这种视图
//InternalResourceView使用其父类AbstractView的render()方法  这个方法我们上面分析过了
//只看下InternalResourceView重写的renderMergedOutputModel()
//发现了吗我们上面已经分析过了  没错就是那个默认创建的视图   此处省略不再分析
protected void renderMergedOutputModel(
                        Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response) throws Exception {

        // Expose the model object as request attributes.
        exposeModelAsRequestAttributes(model, request);

        // Expose helpers as request attributes, if any.
        exposeHelpers(request);

        // Determine the path for the request dispatcher.
        String dispatcherPath = prepareForRendering(request, response);

        // Obtain a RequestDispatcher for the target resource (typically a JSP).
        RequestDispatcher rd = getRequestDispatcher(request, dispatcherPath);
        if (rd == null) {
                throw new ServletException("Could not get RequestDispatcher for [" + getUrl() +
                                "]: Check that the corresponding file exists within your web
application archive!");
        }

        // If already included or response already committed, perform include, else forward.
        if (useInclude(request, response)) {
                response.setContentType(getContentType());
                if (logger.isDebugEnabled()) {
                        logger.debug("Including resource [" + getUrl() + "] in InternalResourceView
'" + getBeanName() + "'");
                }
                rd.include(request, response);
        }

        else {
                // Note: The forwarded resource is supposed to determine the content type itself.
                if (logger.isDebugEnabled()) {
```

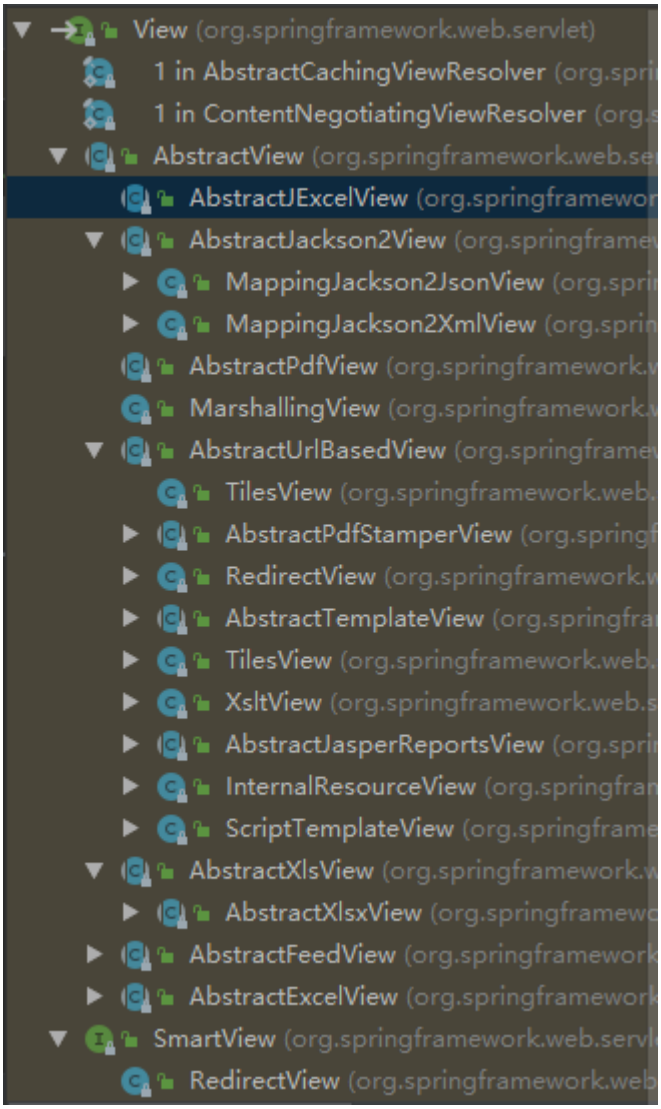
```
                        logger.debug("Forwarding to resource [" + getUrl() + "] in
InternalResourceView '" + getBeanName() + "'");
                }
                rd.forward(request, response);
        }
}
```



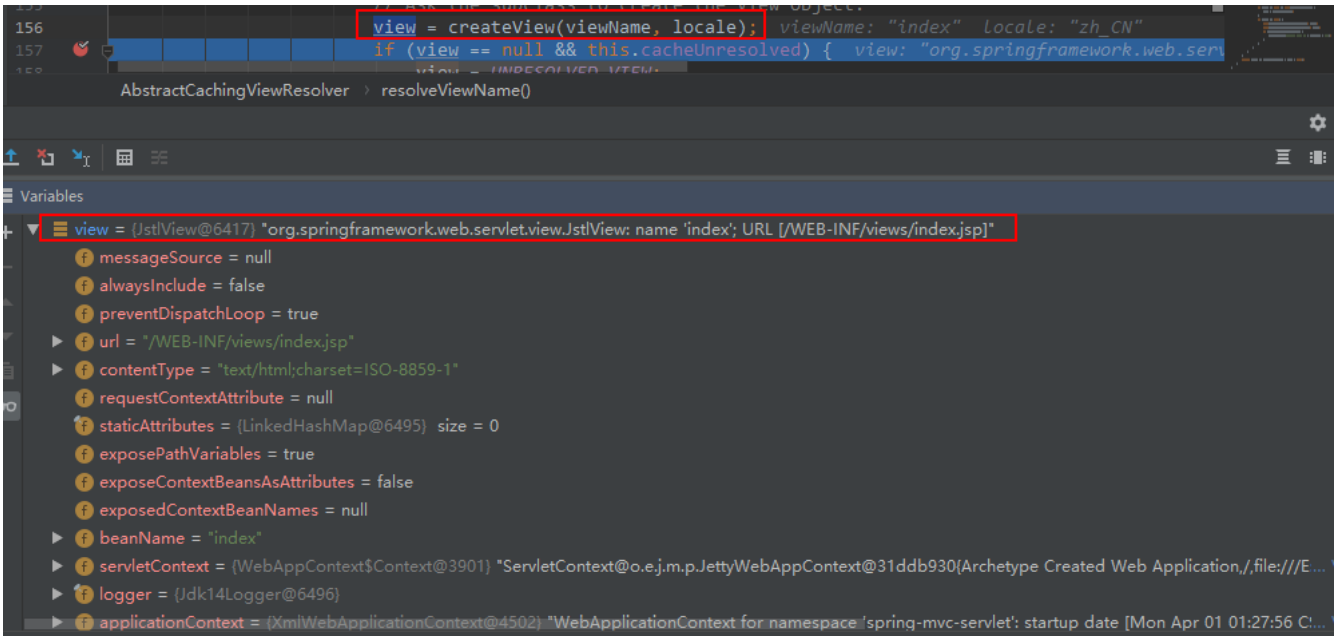
注:本例中的所有viewResolver对象 这个对象就是我们在配置文件中配置的InternalResourceViewResolver



注:ViewResolver接口的主要实现类 本例是InternalResourceView



注:View接口下各种各样的View实现类 本例中主要使用RedirectView 和 InternalResourceView 但是很明显能看出它有pdf，excel，json等View



注:本例中得到的View对象 为何不是InternalResourceView而是JstlView？因为当有JstlView时InternalResourceView会使用它的子类增强版

RequestMappingHandlerAdapter详解

springMvc新推荐的RequestMappingHandlerAdapter解析器其和过时的AnnotationMethodHandlerAdapter处理的思想很类似，只是封装的更好罢了

其入口也是在doDispatch()方法中:

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    //...
    HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());//得到的就是
RequestMappingHandlerAdapter
    //...
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler()); //调用handle()
    //...
}
//调用RequestMappingHandlerAdapter父类的handle()方法
public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object
handler)
            throws Exception {
    //调用RequestMappingHandlerAdapter的handleInternal()方法
    return handleInternal(request, response, (HandlerMethod) handler);
}
//handleInternal()方法
protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

    ModelAndView mav;
    checkRequest(request);

    // Execute invokeHandlerMethod in synchronized block if required.
    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
        else {
            // No HttpSession available -> no mutex necessary
            //执行处理器方法
            mav = invokeHandlerMethod(request, response, handlerMethod);
        }
    }
    else {
        // No synchronization on session demanded at all...
        mav = invokeHandlerMethod(request, response, handlerMethod);
    }

    if (!response.containsHeader(HEADER_CACHE_CONTROL)) {
        if (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {
            applyCacheSeconds(response, this.cacheSecondsForSessionAttributeHandlers);
        }
        else {
```

```
            prepareResponse(response);
        }
    }

    return mav;
}
//invokeHandlerMethod()方法
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

    //还是先封装request和response
    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        //拿到数据绑定器工厂 这个在将来可以直接拿一个数据绑定器的
        WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
        ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);
        //将需要执行的方法做包装  可设置将来方法执行时需要的各种各样的东西  更加符合封装的思想
        ServletInvocableHandlerMethod invocableMethod =
createInvocableHandlerMethod(handlerMethod);
        //放入参数解析器
        invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        //放入返回值处理器
        invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
        //放入数据绑定工厂
        invocableMethod.setDataBinderFactory(binderFactory);
        //放入获取参数名称的工具
        invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);
        //将隐含模型和视图做包装
        ModelAndViewContainer mavContainer = new ModelAndViewContainer();
        mavContainer.addAllAttributes(RequestContextUtils.getInputFlashMap(request));
        //重点方法 初始化模型
        modelFactory.initModel(webRequest, mavContainer, invocableMethod);
        mavContainer.setIgnoreDefaultModelOnRedirect(this.ignoreDefaultModelOnRedirect);
        //Async相关的就是处理异步相关的
        AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(request, response);
        asyncWebRequest.setTimeout(this.asyncRequestTimeout);

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
        asyncManager.setTaskExecutor(this.taskExecutor);
        asyncManager.setAsyncWebRequest(asyncWebRequest);
        asyncManager.registerCallableInterceptors(this.callableInterceptors);
        asyncManager.registerDeferredResultInterceptors(this.deferredResultInterceptors);

        if (asyncManager.hasConcurrentResult()) {
            Object result = asyncManager.getConcurrentResult();
            mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResultContext()[0];
            asyncManager.clearConcurrentResult();
            if (logger.isDebugEnabled()) {
                logger.debug("Found concurrent result value [" + result + "]");
            }
            invocableMethod = invocableMethod.wrapConcurrentResult(result);
        }
        //执行方法以及后续处理
        invocableMethod.invokeAndHandle(webRequest, mavContainer);
        if (asyncManager.isConcurrentHandlingStarted()) {
            return null;
        }
```

```
        return getModelAndView(mavContainer, modelFactory, webRequest);
    }
    finally {
        webRequest.requestCompleted();
    }
}

//initModel() 初始化模型
public void initModel(NativeWebRequest request, ModelAndViewContainer container, HandlerMethod
handlerMethod)
            throws Exception {
    //取出SessionAttributes的session信息
    Map<String, ?> sessionAttributes = this.sessionAttributesHandler.retrieveAttributes(request);
    //session信息放入container
    container.mergeAttributes(sessionAttributes);
    //重要方法 提前执行标准了@ModelAttribute注解的方法
    invokeModelAttributeMethods(request, container);
    //循环执行查找SessionAttribute里的信息 因为session里也是可以放数据的
    for (String name : findSessionAttributeArguments(handlerMethod)) {
        if (!container.containsAttribute(name)) {
            Object value = this.sessionAttributesHandler.retrieveAttribute(request, name);
            if (value == null) {
                throw new HttpSessionRequiredException("Expected session attribute '" + name +
"'", name);
            }
            container.addAttribute(name, value);
        }
    }
}
//invokeModelAttributeMethods() 提前执行标准了@ModelAttribute注解的方法
private void invokeModelAttributeMethods(NativeWebRequest request, ModelAndViewContainer
container)
            throws Exception {

    while (!this.modelMethods.isEmpty()) {
        InvocableHandlerMethod modelMethod = getNextModelMethod(container).getHandlerMethod();
        //拿到@ModelAttribute注解的信息
        ModelAttribute ann = modelMethod.getMethodAnnotation(ModelAttribute.class);
        if (container.containsAttribute(ann.name())) {
            if (!ann.binding()) {
                container.setBindingDisabled(ann.name());
            }
            continue;
        }
        //执行标注了@ModelAttribute注解的方法
        Object returnValue = modelMethod.invokeForRequest(request, container);
        //标注了@ModelAttribute注解的方法如果返回值时void 也是要将其放入container中  类似于以前的处理
@ModelAttribute注解方法时要将其返回值放入隐藏域中是一致的
        if (!modelMethod.isVoid()){
            //返回值类型当@ModelAttribute注解有value值时取value值,没有时取返回值类型首字母小写
            String returnValueName = getNameForReturnValue(returnValue,
modelMethod.getReturnType());
            if (!ann.binding()) {
                container.setBindingDisabled(returnValueName);
            }
            if (!container.containsAttribute(returnValueName)) {
```

```
                //执行结果放入container
                container.addAttribute(returnValueName, returnValue);
            }
        }
    }
}
//invokeForRequest() 执行标注了@ModelAttribute注解的方法
public Object invokeForRequest(NativeWebRequest request, ModelAndViewContainer mavContainer,
        Object... providedArgs) throws Exception {
    //确定执行方法时需要的参数值
    Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
    if (logger.isTraceEnabled()) {
        logger.trace("Invoking '" + ClassUtils.getQualifiedMethodName(getMethod(), getBeanType())
+
                "' with arguments " + Arrays.toString(args));
    }
    //执行方法
    Object returnValue = doInvoke(args);
    if (logger.isTraceEnabled()) {
        logger.trace("Method [" + ClassUtils.getQualifiedMethodName(getMethod(), getBeanType()) +
                "] returned [" + returnValue + "]");
    }
    //返回方法执行后的返回值
    return returnValue;
}

private Object[] getMethodArgumentValues(NativeWebRequest request, ModelAndViewContainer
mavContainer,
            Object... providedArgs) throws Exception {

    //拿到传入的参数的类型
    MethodParameter[] parameters = getMethodParameters();
    Object[] args = new Object[parameters.length];
    for (int i = 0; i < parameters.length; i++) {
        MethodParameter parameter = parameters[i];
        parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
        //解析是否是springMvc提供的原生的API  比如Request等
        args[i] = resolveProvidedArgument(parameter, providedArgs);
        if (args[i] != null) {
            continue;
        }
        //参数解析器开始判断哪个参数解析器可以解析这个参数
        //实际就是调用HandlerMethodArgumentResolverComposite 这个类
        //这里详细说一下HandlerMethodArgumentResolverComposite这个类
        if (this.argumentResolvers.supportsParameter(parameter)) {
            try {
                //解析参数 以自定义参数解析为例 使用ModelAttributeMethodProcessor这个参数处理器
                args[i] = this.argumentResolvers.resolveArgument(
                        parameter, mavContainer, request, this.dataBinderFactory);
                continue;
            }
            catch (Exception ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug(getArgumentResolutionErrorMessage("Failed to resolve", i), ex);
                }
                throw ex;
            }
```

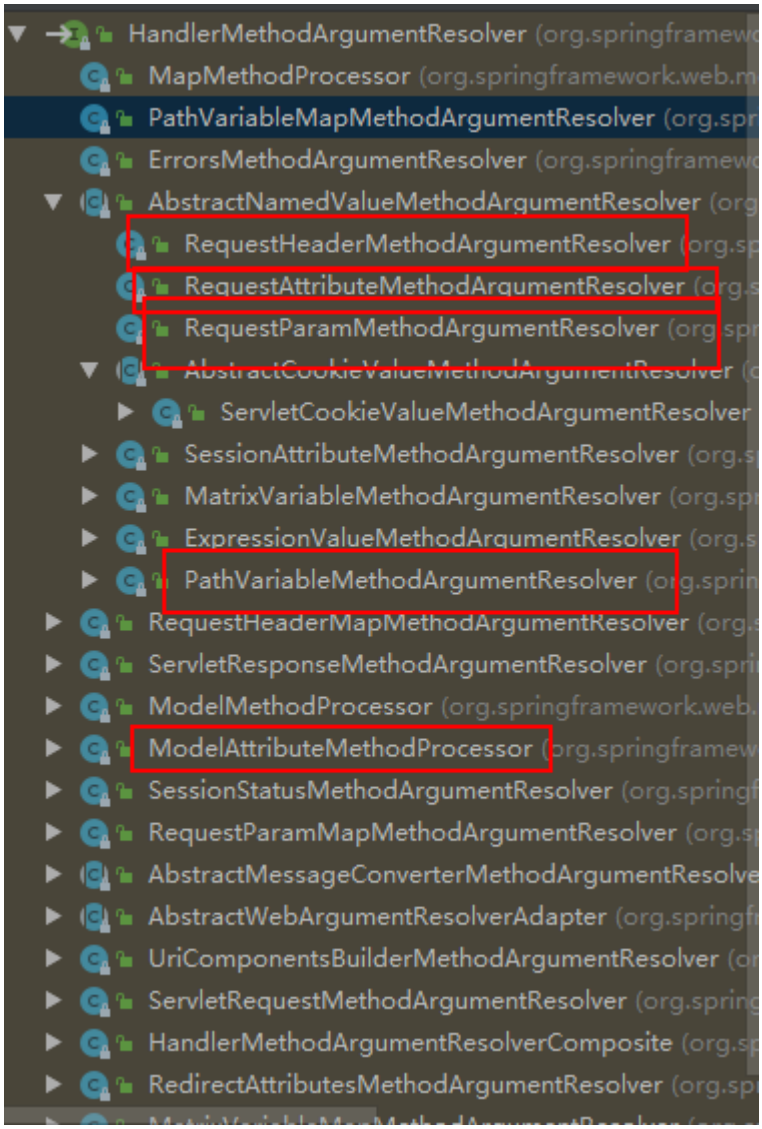
```
        }
        if (args[i] == null) {
            throw new IllegalStateException("Could not resolve method parameter at index " +
                    parameter.getParameterIndex() + " in " +
parameter.getMethod().toGenericString() +
                    ": " + getArgumentResolutionErrorMessage("No suitable resolver for", i));
        }
    }
    //将解析好的参数值返回
    return args;
}
//HandlerMethodArgumentResolverComposite这个类
public class HandlerMethodArgumentResolverComposite implements HandlerMethodArgumentResolver {

    protected final Log logger = LogFactory.getLog(getClass());

    private final List<HandlerMethodArgumentResolver> argumentResolvers =
            new LinkedList<HandlerMethodArgumentResolver>();

    private final Map<MethodParameter, HandlerMethodArgumentResolver> argumentResolverCache =
            new ConcurrentHashMap<MethodParameter, HandlerMethodArgumentResolver>(256);


    /**
     * Add the given {@link HandlerMethodArgumentResolver}.
     */
    public HandlerMethodArgumentResolverComposite addResolver(HandlerMethodArgumentResolver
resolver) {
        this.argumentResolvers.add(resolver);
        return this;
    }

    /**
     * Add the given {@link HandlerMethodArgumentResolver}s.
     * @since 4.3
     */
    public HandlerMethodArgumentResolverComposite addResolvers(HandlerMethodArgumentResolver...
resolvers) {
        if (resolvers != null) {
            for (HandlerMethodArgumentResolver resolver : resolvers) {
                this.argumentResolvers.add(resolver);
            }
        }
        return this;
    }

    /**
     * Add the given {@link HandlerMethodArgumentResolver}s.
     */
    //增加参数解析器
    public HandlerMethodArgumentResolverComposite addResolvers(List<? extends
HandlerMethodArgumentResolver> resolvers) {
        if (resolvers != null) {
            for (HandlerMethodArgumentResolver resolver : resolvers) {
                this.argumentResolvers.add(resolver);
            }
        }
```

```
        return this;
    }

    /**
     * Return a read-only list with the contained resolvers, or an empty list.
     */
    //拿到所有的已经注册了的参数解析器
    public List<HandlerMethodArgumentResolver> getResolvers() {
        return Collections.unmodifiableList(this.argumentResolvers);
    }

    /**
     * Clear the list of configured resolvers.
     * @since 4.3
     */
    public void clear() {
        this.argumentResolvers.clear();
    }


    /**
     * Whether the given {@linkplain MethodParameter method parameter} is supported by any
registered
     * {@link HandlerMethodArgumentResolver}.
     */
    //判断是否支持某个参数类型的解析
    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return (getArgumentResolver(parameter) != null);
    }

    /**
     * Iterate over registered {@link HandlerMethodArgumentResolver}s and invoke the one that
supports it.
     * @throws IllegalStateException if no suitable {@link HandlerMethodArgumentResolver} is
found.
     */
    //根据参数类型解析参数
    @Override
    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {

        HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);
        if (resolver == null) {
            throw new IllegalArgumentException("Unknown parameter type [" +
parameter.getParameterType().getName() + "]");
        }
        return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
    }

    /**
     * Find a registered {@link HandlerMethodArgumentResolver} that supports the given method
parameter.
     */
    //根据参数类型从已经注册了的参数解析器中去循环拿参数解析器 只要匹配上就ok
    private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter parameter) {
        HandlerMethodArgumentResolver result = this.argumentResolverCache.get(parameter);
```

```
        if (result == null) {
            for (HandlerMethodArgumentResolver methodArgumentResolver : this.argumentResolvers) {
                if (logger.isTraceEnabled()) {
                    logger.trace("Testing if argument resolver [" + methodArgumentResolver + "]
supports [" +
                            parameter.getGenericParameterType() + "]");
                }
                if (methodArgumentResolver.supportsParameter(parameter)) {
                    result = methodArgumentResolver;
                    this.argumentResolverCache.put(parameter, result);
                    break;
                }
            }
        }
        return result;
    }

}
//还是说说HandlerMethodArgumentResolver参数解析器这个接口
//所有的参数解析器都要实现这个接口 这个接口就两个方法
public interface HandlerMethodArgumentResolver {
    //是否支持此种类型参数的解析
    boolean supportsParameter(MethodParameter parameter);
    //解析此种类型参数
    Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception;

}
//ModelAttributeMethodProcessor类的resolveArgument()参数处理方法
public final Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,
            NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {

    String name = ModelFactory.getNameForParameter(parameter);
    ModelAttribute ann = parameter.getParameterAnnotation(ModelAttribute.class);
    if (ann != null) {
        mavContainer.setBinding(name, ann.binding());
    }
    //先从mavContainer中取自定义参数的信息 有就直接在模型中取 没有就创建一个
    Object attribute = (mavContainer.containsAttribute(name) ? mavContainer.getModel().get(name) :
            createAttribute(name, parameter, binderFactory, webRequest));

    //创建一个数据绑定器
    WebDataBinder binder = binderFactory.createBinder(webRequest, attribute, name);
    if (binder.getTarget() != null) {
        if (!mavContainer.isBindingDisabled(name)) {
            //数据绑定  将传过来的参数与bean进行绑定 里面有关于数据类型转化器的调用等
            bindRequestParameters(binder, webRequest);
        }
        //数据校验
        validateIfApplicable(binder, parameter);
        if (binder.getBindingResult().hasErrors() && isBindExceptionRequired(binder, parameter)) {
            throw new BindException(binder.getBindingResult());
        }
    }

    // Add resolved attribute and BindingResult at the end of the model
    //数据模型放入mavContainer中
```

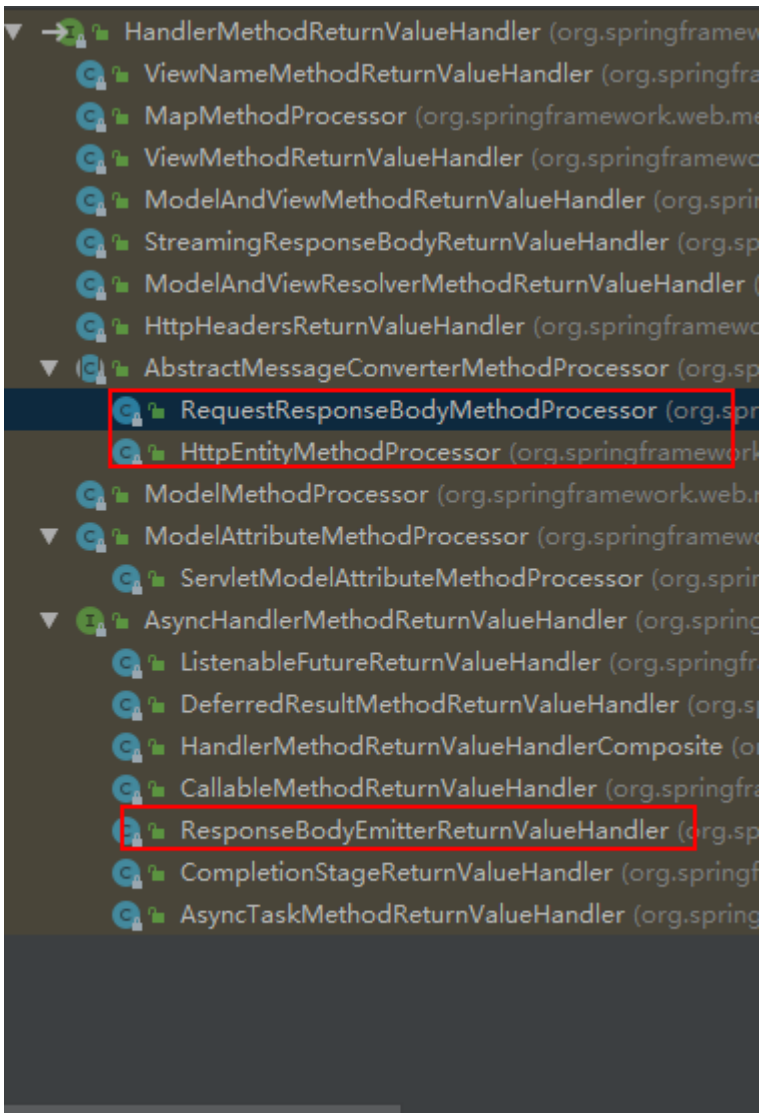
```
    Map<String, Object> bindingResultModel = binder.getBindingResult().getModel();
    mavContainer.removeAttributes(bindingResultModel);
    mavContainer.addAllAttributes(bindingResultModel);

    return binder.convertIfNecessary(binder.getTarget(), parameter.getParameterType(), parameter);
}

//invokeAndHandle() 执行方法以及后续处理
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
            Object... providedArgs) throws Exception {
    //执行方法  上面分析已经分析过了
    Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
    setResponseStatus(webRequest);

    if (returnValue == null) {
        if (isRequestNotModified(webRequest) || getResponseStatus() != null ||
mavContainer.isRequestHandled()) {
            mavContainer.setRequestHandled(true);
            return;
        }
    }
    else if (StringUtils.hasText(getResponseStatusReason())) {
        mavContainer.setRequestHandled(true);
        return;
    }

    mavContainer.setRequestHandled(false);
    try {
        //此处重要  返回值处理器处理返回后的结果
        //这里的返回值处理器其实是和参数处理器的概念是一样的 它主要是对返回值做一些处理
        this.returnValueHandlers.handleReturnValue(
                returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
    }
    catch (Exception ex) {
        if (logger.isTraceEnabled()) {
            logger.trace(getReturnValueHandlingErrorMessage("Error handling return value",
returnValue), ex);
        }
        throw ex;
    }
}
//最后说一下这个RequestMappingHandlerAdapter替代AnnotationMethodHandlerAdapter  感觉就是新版的解析器将模块化玩的淋漓尽致
//想要处理参数,用参数解析器; 想要处理返回值,使用返回值解析器; 想要转化参数,使用参数转化器; 想要校验数据,用数据校验器等等
//模块儿化的设计确实很好  估摸着这种设计还会在后续的版本中加大其他部分的模块化
```

接口实现类的部分展示，这个接口的实现类实在太多了，所有你能想的到的实现参数解析它都实现了



返回值解析器当你的控制器返回的是@ResponseBody,ResponseEntity 这种时其实就是使用返回值解析器做的 但真正调用的是还是数据转化器

```
@ResponseBody
@RequestMapping("/getallajax")
public Collection<Employee> ajaxGetAll(){
    Collection<Employee> all = employeeDao.getAll();
    return all;
}

@RequestMapping("/haha")
public ResponseEntity<String> hahah(){

    MultiValueMap<String, String> headers = new HttpHeaders();
    String body = "<h1>success</h1>";
    headers.add("Set-Cookie", "username=hahahaha");

    return new ResponseEntity<String>(body , headers, HttpStatus.OK);
}
```



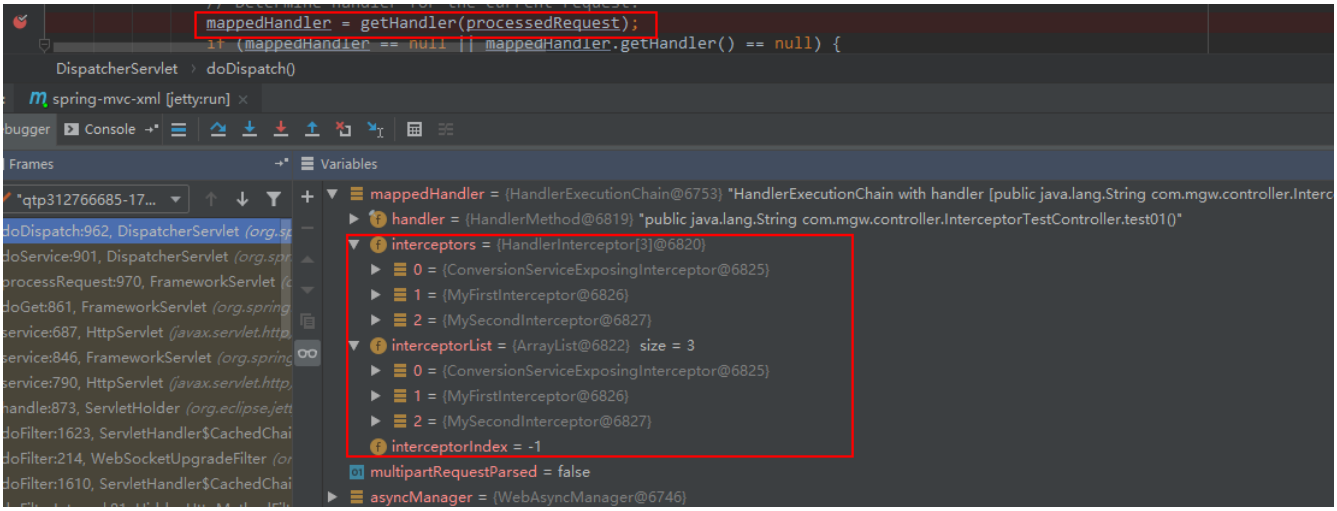
拦截器运行详解

```
//入口方法
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // Determine handler for the current request.
            //前面说过了这个处理器映射器其实是个处理器链  它其中就包含拦截器
            mappedHandler = getHandler(processedRequest);
```

```
            if (mappedHandler == null || mappedHandler.getHandler() == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // Determine handler adapter for the current request.
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

            // Process last-modified header, if supported by the handler.
            String method = request.getMethod();
            boolean isGet = "GET".equals(method);
            if (isGet || "HEAD".equals(method)) {
                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                if (logger.isDebugEnabled()) {
                    logger.debug("Last-Modified value for [" + getRequestUri(request) + "] is: " +
lastModified);
                }
                if (new ServletWebRequest(request, response).checkNotModified(lastModified) &&
isGet) {
                    return;
                }
            }
            //在适配器执行方法之前，执行所有拦截器的preHandle()方法
            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                //一旦有一个拦截器的preHandle()方法返回的是false  直接返回 后续拦截器 处理方法都不再执行
                return;
            }

            // Actually invoke the handler.
            //执行目标方法
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }

            applyDefaultViewName(processedRequest, mv);
            //目标方法执行只要没异常，那么所有拦截器的postHandle()方法就可以执行到
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
            dispatchException = ex;
        }
        catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler methods as well,
            // making them available for @ExceptionHandler methods and other scenarios.
            dispatchException = new NestedServletException("Handler dispatch failed", err);
        }
        //注意这里 不管目标方法是否异常这个是必然执行的 因为就算发生错误  错误页面也是需要渲染的嘛
        //还有这个方法如果异常 那么都会调用拦截器的afterCompletion()方法会在下面的catch块中执行
        processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
    }
    catch (Exception ex) {
        //任何时候发生异常 都会调用拦截器的afterCompletion() 方法
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
    catch (Throwable err) {
```

```
        triggerAfterCompletion(processedRequest, response, mappedHandler,
                new NestedServletException("Handler processing failed", err));
    }
    finally {
        if (asyncManager.isConcurrentHandlingStarted()) {
            // Instead of postHandle and afterCompletion
            if (mappedHandler != null) {
                mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
            }
        }
        else {
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
}
//所有拦截器执行preHandle()
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws Exception
{
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = 0; i < interceptors.length; i++) {
            HandlerInterceptor interceptor = interceptors[i];
            //一旦拦截器的preHandle()执行为false 直接返回false 后续拦截器的postHandle()方法不再执行
            if (!interceptor.preHandle(request, response, this.handler)) {
                //执行已经放行的拦截器的afterCompletion() 方法
                triggerAfterCompletion(request, response, null);
                return false;
            }
            //记录一下已经放行的拦截器索引值
            this.interceptorIndex = i;
        }
    }
    return true;
}
//执行已经放行的拦截器的afterCompletion() 方法
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, Exception
ex)
            throws Exception {
    //拿到已经放行的拦截器索引值
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        //反索引调用  此处就是我们所知的afterCompletion()方法是反顺序调用的
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
            try {
                //执行afterCompletion()方法
                interceptor.afterCompletion(request, response, this.handler, ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
            }
        }
    }
}
```

```
//所有拦截器的postHandle()方法执行
void applyPostHandle(HttpServletRequest request, HttpServletResponse response, ModelAndView mv)
throws Exception {
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        //postHandle()方法反顺序执行的
        for (int i = interceptors.length - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
            //执行postHandle()方法
            interceptor.postHandle(request, response, this.handler, mv);
        }
    }
}
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
            HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) throws
Exception {

    boolean errorView = false;

    if (exception != null) {
        if (exception instanceof ModelAndViewDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered", exception);
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
        }
        else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
            mv = processHandlerException(request, response, handler, exception);
            errorView = (mv != null);
        }
    }

    // Did the handler return a view to render?
    if (mv != null && !mv.wasCleared()) {
        render(mv, request, response);
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    }
    else {
        if (logger.isDebugEnabled()) {
            logger.debug("Null ModelAndView returned to DispatcherServlet with name '" +
getServletName() +
                    "': assuming HandlerAdapter completed request handling");
        }
    }

    if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Concurrent handling started during a forward
        return;
    }

    if (mappedHandler != null) {
        //页面渲染完毕后也会执行拦截器的afterCompletion()方法
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}
```


mappedHandler = getHandler(processedRequest);
if (mappedHandler == null || mappedHandler.getHandler() == null) {
DispatcherServlet › doDispatch()
spring-mvc-xml [jetty:run]
Console
Frames
Variables
"qtp312766685-17...
doDispatch:962, DispatcherServlet
doService:901, DispatcherServlet
processRequest:970, FrameworkServlet
doGet:861, FrameworkServlet
service:687, HttpServlet
service:846, FrameworkServlet
service:790, HttpServlet
handle:873, ServletHolder
doFilter:1623, ServletHandler$CachedChai
doFilter:214, WebSocketUpgradeFilter
doFilter:1610, ServletHandler$CachedChai
mappedHandler = {HandlerExecutionChain@6753} "HandlerExecutionChain with handler [public java.lang.String com.mgw.controller.Interc
handler = {HandlerMethod@6819} "public java.lang.String com.mgw.controller.InterceptorTestController.test01()"
interceptors = {HandlerInterceptor[3]@6820}
0 = {ConversionServiceExposingInterceptor@6825}
1 = {MyFirstInterceptor@6826}
2 = {MySecondInterceptor@6827}
interceptorList = {ArrayList@6822} size = 3
0 = {ConversionServiceExposingInterceptor@6825}
1 = {MyFirstInterceptor@6826}
2 = {MySecondInterceptor@6827}
interceptorIndex = -1
multipartRequestParsed = false
asyncManager = {WebAsyncManager@6746}

注:处理器链